the art of UNIT TESTING

with examples in JavaScript

THIRD EDITION

MANNING

Roy Osherove

MEAP Edition
Manning Early Access Program

The Art of Unit Testing

with examples in JavaScript
Third Edition
Version 7

welcome

Thanks for purchasing the MEAP for *The Art of Unit Testing 3rd Edition*, with Examples in Javascript.

This book has been written to take most anyone from never doing unit testing in Javascript to being able to write real world unit tests on real projects with real code.

We will begin at the very basics of what makes good unit tests, and move to mocks, stubs, async work and refactoring. We're using Jest as our test framework but that shouldn't stop you from using any other unit test framework for your purposes.

The book assumes that you're already familiar with Javasctipt ES2015 and that's it. It should fit well whether you're a backend or frontend developer because the examples will try to be as agnostic as possible to the various frameworks out there such as Express, React or Angular, and instead focus on the patterns underlying the code structures, so that you should be able to approach most any task with the knowledge that you've seen this pattern before, be it with a slightly different disguise.

—Roy Osherove

brief contents

Part 1
Getting started

This part of the book covers the basics of unit testing.

In chapter 1, I'll define what a *unit* is and what "good" unit testing means, and I'll compare unit testing with integration testing. Then we'll look at test-driven development and its role in relation to unit testing.

You'll take a stab at writing your first unit test using *Jest* (A very common JavaScript test framework) in chapter 2. You'll get to know *Jest*'s basic API, how to assert things, and how to execute tests continuously.

1 The basics of unit testing

This chapter covers

- Defining entry points & exit points
- Defining a unit of work & unit tests
- Contrasting unit testing with integration testing
- Exploring a simple unit testing example
- Understanding test-driven development

SOME ASSUMPTIONS BEFORE WE BEGIN:

You already know basic usage of NODE.JS and node package manager (NPM). You know how to use GIT source control. You've seen github.com before and you know how to clone a repository from there. You've written code in JavaScript , either backend or frontend code, and you can understand basic ES6 JavaScript code examples

CAN YOU USE THIS BOOK FOR NON JAVASCRIPT USE?

Yes. The previous editions of this book were in C#. I've foun that about 80% of the patterns there have transferred over quite easily. So you should be able to read this book eve if you come from Java, .NET, Python, Ruby or other languages. The patterns are just patterns. The language being used is there just so we can demonstrate those patterns. But they are not language specific.

1.1 The first step

There’s always a first step: the first time you wrote a program, the first time you failed a project, and the first time you succeeded in what you were trying to accomplish. You never forget your first time, and I hope you won’t forget your first tests.

- You may have come across them in some form; Some of your favorite open source projects come with bundled up “test” folders, you have them in your own project at work.
- You might have already written a few tests yourself, and you may even remember them as being bad, awkward, slow, or unmaintainable. Even worse, you might have felt they were useless and a waste of time. (Many people sadly do.)
- On a more upbeat note, you may have had a great first experience with unit tests, and you’re reading this to see what more you might be missing.

This chapter will first analyze the “classic” definition of a unit test and compare it to the concept of integration testing. This distinction is confusing to many, but is very important to learn because, as we’ll learn later in the book, separating unit tests from other types of tests can be crucial to having high confidence in your tests when they fail or pass.

We’ll also discuss the pros and cons of unit testing versus integration testing and develop a better definition of what might be a “good” unit test. We’ll finish with a look at test-driven development, because it’s often associated with unit testing, but is a separate skill that I highly recommend giving a chance (it’s not the main topic of this book though). Throughout the chapter, I’ll also touch on concepts that are explained more thoroughly elsewhere in the book.

First, let’s define what a unit test should be.

1.2 Defining unit testing, step by step

Unit testing isn’t a new concept in software development. It’s been floating around since the early days of the Smalltalk programming language in the 1970s, and it proves itself time and time again as one of the best ways a developer can improve code quality while gaining a deeper understanding of the functional requirements of a module, class or function.

Kent Beck introduced the concept of unit testing in Smalltalk, and it has carried on into many other programming languages, making unit testing an extremely useful practice in software programming.

To see what we *don’t want to* use as our definition of unit testing, let’s first look to Wikipedia as a starting point. I’ll use this one because there are many important pats missing for me but it is largely “accepted” by many for lack of other good definitions.. It’ll be slowly evolving throughout this chapter, with the final definition appearing in section 1.8.

DEFINITION 1.0 (WIKIPEDIA) Unit tests are typically automated tests written and run by software developers to ensure that a section of an application (known as the "unit") meets its design and behaves as intended. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method.

The thing you'll write tests for is called the "SUT".

DEFINITION SUT stands for subject, *system or suite under test*, and some people like to use CUT (*component, module or class under test* or *code under test*). When you test something, you refer to the thing you're testing as the SUT.

Let's talk about the word "unit" in unit testing. To me, a *unit* stands for "unit of work" or a "use case" inside the system. A unit of work has a beginning and an end. I call these *entry points* and *exit points*. A simple example of a unit of work, as we'll soon see, is a function that calculates something and returns a value. But that function could also use other functions , other modules and other components in the calculation process, which means the unit of work (from entry point to exit point), now spans more than just a function.

Definition : Unit of Work

A *unit of work* is the sum of actions that take place between the invocation of an *entry point* up until a noticeable end result through one or more *exit points*. The *entry point* is the thing we trigger. Given a publicly visible function, for example:

The function's body is all or part of the unit of work.
The function's declaration and signature are the entry point into the body.
The resulting outputs or behaviors of the function are its exit points.

1.3 Entry Points & Exit Points

A unit of work always has an entry point and one or more exit points.

Figure 1.1 shows a simple diagram of how I think about Units of Work:

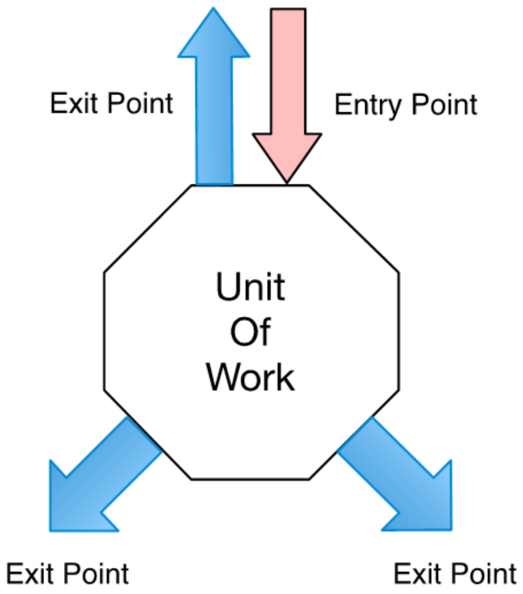


Figure 1.1: A Unit of work has entry points and exit points.

A *unit of work* can be a single function, multiple functions, multiple functions or even multiple modules or components. But it always has an entry point which we can trigger from the outside (via tests or other production code), and it always ends up doing something useful. If it doesn't do anything useful, we might as well remove it from our codebase.

What's useful? Something publicly noticeable happens in the code: A return value, state change or calling an external party. Those noticeable behaviors are what I call exit points. Listing 1.1 will show a simple version of a unit of work.

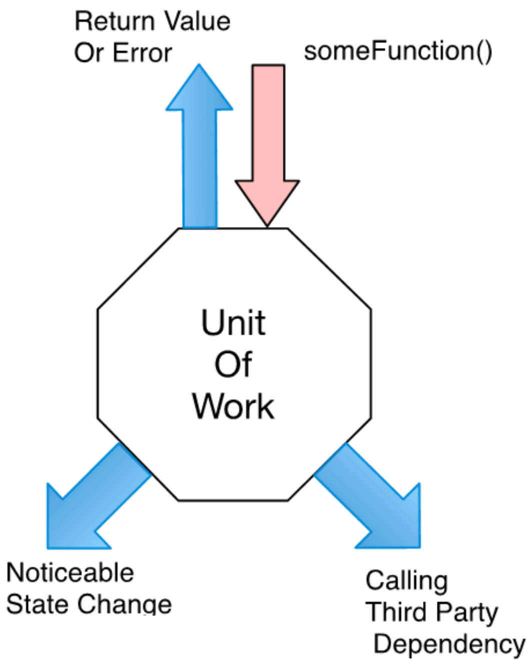


Figure 1.2: Types of Exit Points

Why "exit point"?

I got asked by an early reviewer of this book why I choise to use "exit point" and not something like "behavior". My thinking is that behaviors can also be purely internal. We're looking for externally visible behaviors from the caller. That distinction might be difficult to differentiate during "live action" coding. Also, "exit point" has the nice connotation to it that suggests we are leaving the context of a unit of work and going back to the test context. Behaviors might be a bit more fluid than that. That said, I'm not sure. Perhaps this will change in the 4th edition of this book…

Listing 1.1 shows a quick code example of a simple unit of work.

Listing 1.1: A simple function that we'd like to test. /ch1/number-parser.js

```
const sum = (numbers) => {
  const [a, b] = numbers.split(',');
  const result = Number.parseInt(a, 10) +
        Number.parseInt(b, 10);
  return result;
};

module.exports.sum = sum;
```

Quick note on the JavaScript version used in this book:

I've chosen to use Node.js 12.8 with Plain ES6 JavaScript along with *JSDoc* style comments. The module system I will use is CommonJS to keep things simple. Perhaps in a future edition I'll start using ES modules (.mjs files) but for now, and for the rest of this book, CommonJS will do. It doesn't really matter for the patterns in this book anyway.

You should be able to easily extrapolate the techniques to whatever JavaScript stack you're currently working with, and apply them whether you're working with TypeScript, Plain JS, ES Modules, Backend or Frontend, Angular or React. It shouldn't matter.

Getting the code for this chapter

You can download all the code samples that are shown in this book on *github*. You can find the repository at https://github.com/royosherove/aout3-samples. Make sure you have node 12.8 or higher installed and run "npm install" followed by "npm run ch[chapter number]". For this chapter you would run "npm run ch1". This will run all the tests for this chapter so you van see their outputs.

This unit of work is completely encompassed in a single function. The function is both the entry point, and because its end result returns a value, it also acts as the exit point. We get the end result in the same place we trigger the unit of work, so we can describe it such that the entry point is also the exit point.

If we drew this function as a unit of work, it would look something like figure 1.3:

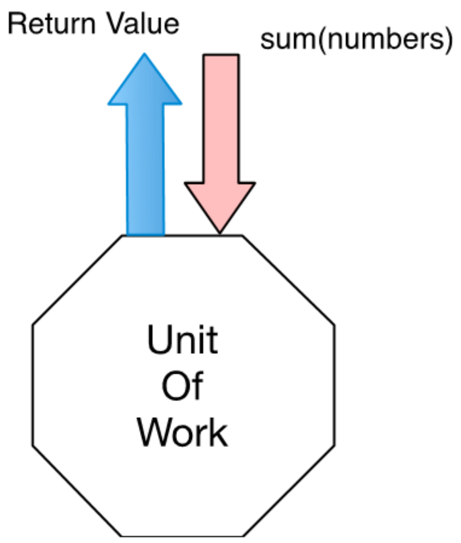


Figure 1.3: A function that has the same entry point as the exit point.

I used "sum(numbers" as the entry point and not "numbers" because the entry point is the function signature. The parameters are the context or input given through the entry point.

Here's a variation of this idea. Take a look at listing 1.2.

Listing 1.2: A Unit of work has entry points and exit points (number-parser2.js)

```
let total = 0;

const totalSoFar = () => {
  return total;
};

const sum = (numbers) => {
  const [a, b] = numbers.split(',');
  const result = Number.parseInt(a, 10) +
            Number.parseInt(b, 10);
  total += result;
  return result;
};

module.exports = {
  sum,
  totalSoFar
};
```

This new version of `sum` has *two* exit points. It does two things:

- It returns a value
- it has new functionality: it has a running total of all the sums. It sets the state of the module in a way that is noticeable (via `totalSoFar()`) from the caller of the entry point.

Figure 1.4 shows how I would draw that:

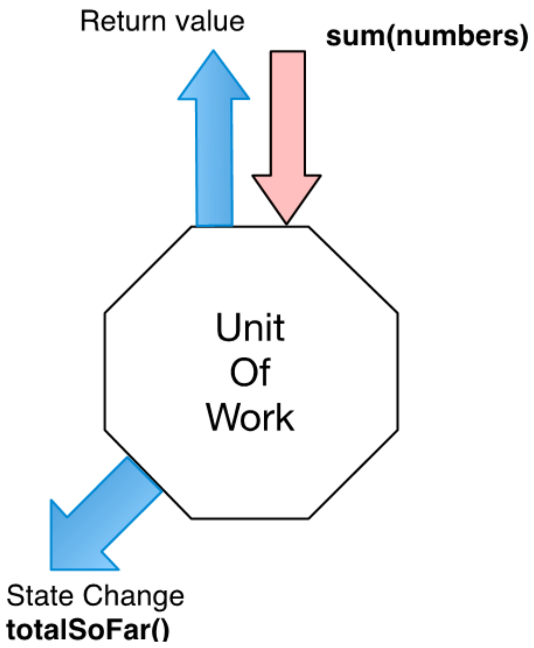


Figure 1.4: A unit of work with two exit points.

You can think of these two exit points as two different *paths*, or *requirements* from the same unit of work, because they indeed *are* two different *useful* things the code is expected to do.

It also means I'd be very likely to write two different unit tests here: one for each exit point. Very soon we'll do exactly that.

What about `totalSoFar()`? Is this also an entry point? Yes, it could be. IN a separate test. I could write a test that proves that calling `totalSoFar()` without triggering prior to that call returns 0. Which would make it its own little unit of work. And would be perfectly fine. Often one unit of work (`sum()`) can be made of smaller units. This is where we can start to see how the scope of our tests can change and mutate, but we can still define it with entry points and exit points.

Entry points are always where the test triggers the unit of work. You can have multiple entry points into a unit of work, each used by a different set of tests.

A side note on design

In terms of design, you can also think of it like this. There are two main types of actions: "Query" actions, and "Command" functions. Query actions don't change stuff, they just return values. Command actions change stuff but don't return values.

We often combine the two but there are many cases where separating them might be a better design choice. This book isn't primarily about design, but I urge you to read more about the concept of command-query separation over at Martin Fowler's website: https://martinfowler.com/bliki/CommandQuerySeparation.html

Exit points often signify requirements and new tests, and vice versa

Exit points are end results of a unit of work. For Unit Tests, I usually write at least one separate test, with its own readable name, for each exit point. I could then add more tests with variations on inputs, all using the same entry point, to gain more confidence.

Integration tests, which will be discussed later on in this chapter and in the book, will usually include multiple end results since it could be impossible to separate code paths at those levels. That's also one of the reasons integration tests are harder to debug, get up and running, and maintain: they do much more than unit tests, as we'll soon see.

Here's a third version of this function (located in `ch1/number-parser3.js`):

Listing 1.3: Adding a logger call to the function (number-parser3.js)

```
const winston = require('winston');
let total = 0;

const totalSoFar = () => {
  return total;
};

const makeLogger = () => {
  return winston
    .createLogger({
      level: 'info',
      transports: new winston.transports.Console()
    });
};

const logger = makeLogger();

const sum = (numbers) => {
  const [a, b] = numbers.split(',');
  logger.info(
    'this is a very important log output',
    { firstNumWas: a, secondNumWas: b });

  const result = Number.parseInt(a, 10) + Number.parseInt(b, 10);
  total += result;
  return result;
};

module.exports = {
  totalSoFar,
  sum
};
```

You can see there's a new *exit point/requirement/end result* in the function. It logs something to an external entity. It could be a file, or the console, or a database. We don't know, and we don't care.

This is the third type of an exit point: *calling a third party*. I also like to call it "calling a *dependency*".

A Dependency (3rd party)

A dependency is something we don't have full control over during a unit test. Or something that, to try to control in a test, would make our lives miserable to get up and running, maintain, keep the test consistent, or running fast. Some examples would include: loggers that write to files, things that talk to the network, code controlled by other teams that we cannot change, components that due to the way they function take a very long time (calculations, threads, database access) and more. The rule of thumb is: "If I can fully and easily control what its doing, and it runs in memory, and its fast, it's not a dependency". There are always exceptions to the rule, but this should get you through 80% of the cases at least.

Here's how I'd draw it with all three exit points, and two entry points together:

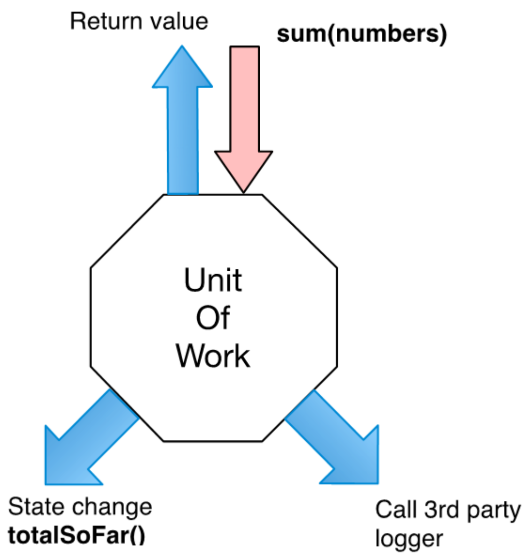


Figure 1.5: showing all three exit points from the same function.

At this point we're still just a function-sized unit of work. The entry point is the function call, but now we have three possible paths, or exit points, that do something useful that the caller can verify publicly.

Here's where it gets interesting: It's a good idea to have a *separate test per exit point*. This will make the tests more readable, simpler and easier to debug or change without affecting other outcomes.

1.4 Exit Point Types

We've seen that we have three different types of end results:

- The invoked function returns a useful value (not undefined). If this was in a more static language such as Java or C# we'd say it is a public, non void function.
- There's a *noticeable* change to the state or behavior of the system before and after invocation that can be determined without interrogating private state. (In our case the `wasCalled()` function returns a different value after the state change.)
- There's a callout to a third-party system over which the test has no control. That third-party system doesn't return any value, or that value is ignored. (Example: calling a third-party logging system that was not written by you and you don't control its source code.)

XUnit Test Patterns' Definition of Entry & Exit Points

The book *XUnit Test Patterns* discusses the notion of *direct inputs and outputs,* and *indirect inputs and outputs*. *Direct inputs* are what I like to call entry points. That book called it "using the front door" of a component. Indirect outputs in that book can be thought of as the other two types of exit points I mentioned (state change and calling a third party). Both versions of these ideas have evolved in parallel, but the idea of "unit of work" only appears in this book. Unit of work , coupled with "entry" and "exit" points makes much more sense to me that using "direct and indirect inputs and outputs". Consider this a stylistic choice on how to teach the concept of test scope. You can find more about xunit test patterns at xunitpatterns.com

- Let's see how the idea of entry and exit points affects the definition of a unit test.

UPDATED DEFINITION 1.1 A *unit test* is a piece of code that invokes a unit of work and **checks one specific exit point** as an end result of that unit of work. If the assumptions on the end result turn out to be wrong, the unit test has failed. A unit test's **scope can span as little as a function or as much as multiple modules or components depending on how many functions and modules are used between the entry point and the exit point.**

1.5 Different Exit Points, Different Techniques

Why am I spending so much time just talking about types of exit points? Because not only is it a great idea to separate the tests per exit point, but also each type of exit point might require a different technique to test successfully.

- Return-value based exit points (direct outputs per "xunit patterns") of all the exit point types, should be the easiest to test. You trigger an entry point, you get something back, you check the value you get back.

- State based tests (indirect outputs) require a little more gymnastics usually. You call something, then you do *another call to* check *something else* (or call the previous thing again) to see if everything went according to plan.

  In a third-party situation (indirect outputs) we have the most hoops to jump through. We haven't discussed the idea yet, that's where we're forced to use things like *mock objects* in our tests so we can replace the external system with one we can control and interrogate in our tests. I'll cover this idea deeply later in the book.

Which exit points make the most problems?

As a rule of thumb, I try to keep most of my tests either return value based or state based tests. I try to avoid mock-object based tests if I can, and usually I can. As a result I usually have no more than perhaps 5% of my tests using mock objects for verification. Those types of tests complicate things and make maintainability more difficult. Sometimes there's no escape though, and we'll discuss them as we proceed into the next chapters.

1.6 A Test from Scratch

Let's go back to the first, simplest version of the code (listing 1.3), and try to test it, shall we? *If* we were to try to a write a test for this, what would it look like?

Let's take the visual approach first with figure 1.6:

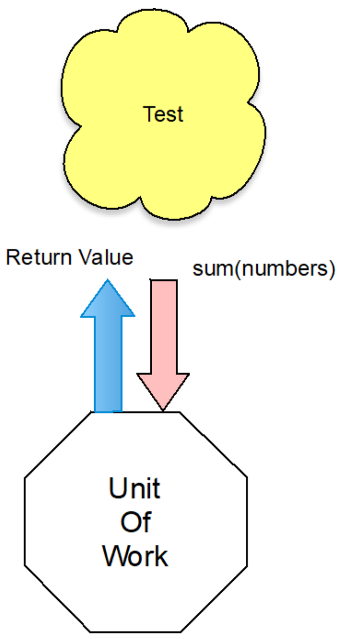


Figure 1.6 A visual view of our test.

Our entry point is *sum* with an input of a *string* called *numbers. Sum* is also our exit point since we will get a return value back from it, and check its value. It's possible to write an automated unit test without using a test framework. In fact, because developers have gotten more into the habit of automating their testing, I've seen plenty of them doing this before discovering test frameworks. In this section, I'll show what writing such a test without a framework can look like, so that you can contrast this with using a framework in chapter 2.

So, let's assume test frameworks don't exist, (or we don't know that they do). We decide to write our own little automated test from scratch. Listing 1.4 shows a very naïve example of testing our own code with plain javascript:

Listing 1.4 A very naïve test against sum() (ch1/custom-test-phase1.js)

```
const { sum } = require('./number-parser');

const parserTest = () => {
  try {
    const result = sum('1,2');
    if (result === 3) {
      console.log('parserTest example 1 PASSED');
    } else {
      throw new Error(`parserTest: expected 3 but was ${result}`);
    }
  } catch (e) {
    console.error(e.stack);
  }
};

parserTest();
```

Yes, this code is not lovely. But it's good enough to explain the point which is: "how do tests work?" order to run this we can simply add an entry under package.json's "scripts" entry under "test" to execute "node mytest.js" and then execute "npm test" in the command line. Listing 1.5 shows this.

Listing 1.5 the beginning of our package.json file

```
{
  "name": "aout3-samples",
  "version": "1.0.0",
  "description": "Code Samples for Art of Unit Testing 3rd Edition",
  "main": "index.js",
  "scripts": {
    "test": "node ./ch1/custom-test-phase1.js",
  },
.
.
.
```

The test method invokes the *production module* (the *SUT*) and then checks the returned value. If it's not what's expected, the test method writes to the console an error and a stack trace. It also catches any exception and writes it to the console.

It’s the test method’s responsibility to catch any exceptions that occur and write them to the console, so that they don’t interfere with the running of subsequent methods. When we use a test framework, that’s usually handled for us automatically.

Obviously, this is an ad hoc way of writing such a test. If you were writing multiple tests like this, you might want to have a generic `“test” or “check”` method that all tests could use, which would format the errors consistently. You could also add special helper methods that would help check on things like null objects, empty strings, and so on, so that you don’t need to write the same long lines of code in many tests.

The following listing shows what this test would look like with a slightly more generic `check` and `assertEquals` functions.

Listing 1.6 Using a more generic implementation of the `Check` method

```
const { sum } = require('./number-parser');

const assertEquals = (expected, actual) => {
  if (actual !== expected) {
    throw new Error(`Expected ${expected} but was ${actual}`);
  }
};

const check = (name, implementation) => {
  try {
    implementation();
    console.log(`${name} passed`);
  } catch (e) {
    console.error(`${name} FAILED`, e.stack);
  }
};

check('sum with 2 numbers should sum them up', () => {
  const result = sum('1,2');
  assertEquals(3, result);
});

check('sum with multiple digit numbers should sum them up', () => {
  const result = sum('10,20');
  assertEquals(30, result);
});
```

I’ve now created two helper methods: `assertEquals`, that removes boilerplate code for writing to the console, or throwing errors, and `check`, that takes a string for the name of the test, and a callback to the implementation. It then takes care of catching any test errors and writing them to the console, and also reporting on the status of the test.

Built in Asserts.

It’s important to note that we didn’t really need to write our own asserts. We could have easily used node.js’ built in *assert* functions, which were originally built for internal use in testing node.js itself By importing it with

```
const assert = require('assert');
```

However, I'm trying to show the underlying simplicity of the concept, so, we'll avoid that. You can find more info about Node.js's assert module at https://nodejs.org/api/assert.html

Notice how the tests are easier to read and faster to write with just a couple of helper methods. Unit testing frameworks such as *jest* can provide even more generic helper methods like this, so tests are written more easily. I'll talk about that in chapter 2. Let's talk a bit about the main subject of this book: "good" unit tests.

1.7 Characteristics of a good unit test

No matter what programming language you're using, one of the most difficult aspects of defining a unit test is defining what's meant by a "good" one. Of course, good is relative, and can change whenever we learn something new about coding. It may seem obvious but it really isn't. I need to explain *why* we need to write better tests.

Being able to understand what a unit of work is isn't enough.

From my own experience over many companies and teams over the years, most people who try to unit test their code either give up at some point or don't actually perform unit tests. They waste a lot of time writing problematic tests and then give up when they have to waste a lot of time maintaining them or worse, not trusting their results.

The way I see it, there's no point in writing a bad unit test, unless you're in the process of learning how to write a good one. There are more downsides that upsides to writing bad tests. The downsides can range from wasting tie debugging tests that are buggy, wasting time writing tests that bring no benefit, wasting time trying to understand unreadable tests, or wasting time writing them only to delete them a few months later. There's also a huge issue with maintainability of bad tests, and how they affect the maintainability of production code. In effect, bad tests can actually *slow* down your development speed, not only for writing test code, but also for writing production code. I touch on all these things later in the book.

By defining what a good unit test is, you can make sure you don't start off on a path that will be too hard to fix later on when the code becomes a nightmare.

We'll also define other forms of tests (component, end to end and more, later in the book.

1.7.1 So, what's a "good" unit test?

All great automated tests (not just unit tests) I've seen have had the following properties:

- It should be easy to read and understand the intent of the test author.
- It should be easy to read and write
- It should be automated and repeatable.
- It should be useful and provide actionable results.
- Anyone should be able to run it at the push of a button.
- When it fails, it should be easy to detect what was expected and determine how to

pinpoint the problem.

Those properties can apply to any type of test.

Great *unit tests should* also exhibit the following properties:

- It should run quickly.
- It should be consistent in its results (it always returns the same result if you don't change anything between runs).
- It should have full control of the code under test.
- It should be fully isolated (runs independently of other tests).
- It should run in memory without requiring system files, networks, databases It should be as synchronous and linear as possible when it makes sense. (no parallel threads if we can help it)

Here are some questions that came up from early reviewers of this chapter:

- Are in-memory databases OK in a unit testing context?
- in memory databases are OK but I make sure they live up to a few simple rules in unit tests: They should be able to load fast. They should be fully configured from code, their data should be the minimum required to prove the point of the test, and a database instance is never shared between tests.
- Does this mean that I can't unit test javascript which interacts with a DOM, browser or node APIs?
- Not necessarily. We can use a DOM abstraction such as JSDom to run purely in memory.. We can also use Headless Chrome or FireFox. It's not ideal, but it's as close to running purely in memory but providing us the required harness our code requires
- Synchronous? What about Promises, and async/await?
- From a unit testing point of view, we should find a way to make the processes happening through a single path in a unit of work, from entry point to exit point, as synchronous as possible. That might mean triggering callbacks directly from the test, or it might mean faking the time in the test to the point where the test does not need to wait for any async code to finish executing, unless there's no other choice (at that point it might not be a unit test, but the situations can vary quite a bit). I'll touch on this in later chapters. **REVIEWERS: I'd like examples of code that is async that you'd want to throw at me to figure out how to test it in a non async way)**

1.7.2 A simple checklist

Many people confuse the act of testing their software with the concept of a unit test. To start off, ask yourself the following questions about the tests you've written/executed up to now:

- Can I run and get results from a test I wrote two weeks or months or years ago?
- Can any member of my team run and get results from tests I wrote two months ago?
- Can I run all the tests I've written in no more than a few minutes?
- Can I run all the tests I've written at the push of a button?

- Do the tests I wrote always report the same outcome (given no code changes on my team)?
- Do my tests pass when there are bugs in another team's code?
- Do my test show the same result when run on different machines or environments?
- Do my tests stop working if there's no database, network or deployment?
- If I delete, move or change one test, do other tests remain unaffected?
- Can I easily set up the needed state for a test without relying on outside resources like a database or a 3rd party, with very little code?

If you've answered no to any of these questions, there's a high probability that what you're implementing either isn't fully automated or it isn't a unit test. It's definitely *some* kind of test, and it's *as* important as a unit test, but it has drawbacks compared to tests that would let you answer yes to all of those questions.

"What was I doing until now?" you might ask. You've been doing integration testing.

1.8 Integration tests

I consider integration tests as any tests don't live up to one or more of the conditions outlined above under "great unit tests". For example, if the test uses the real network, the real rest APIs, real system time, the real filesystem, or a real database, it has stepped into the realm of integration testing.

If a test doesn't have control of the system time, for example, and it uses the current `new Date()` in the test code, then every time the test executes, it's essentially a different test because it uses a different time. It's no longer consistent.

That's not a bad thing per se. I think integration tests are important counterparts to unit tests, but they should be separated from them to achieve a feeling of "safe green zone," which is discussed later in this book.

If a test uses the real database, for example, then it's no longer only running in memory, in that its actions are harder to erase than when using only in-memory fake data. The test will also run longer, and we won't be able to control how long data access takes easily. Unit tests should be *fast*. Integration tests are usually much slower. When you start having hundreds of tests, every half-second counts.

Integration tests increase the risk of another problem: testing too many things at once.

What happens when your car breaks down? How do you learn what the problem is, let alone fix it? An engine consists of many subsystems working together, each relying on the others to help produce the final result: a moving car. If the car stops moving, the fault could be with any of these subsystems—or more than one. It's the integration of those subsystems (or layers) that makes the car move. You could think of the car's movement as the ultimate integration test of these parts as the car goes down the road. If the test fails, all the parts fail together; if it succeeds, all the parts succeed.

The same thing happens in software. The way most developers test their functionality is through the final functionality of the app or a rest API or a UI. Clicking some button triggers a series of events—functions, modules and components working together to produce the final result. If the test fails, all of these software components fail as a team, and it can be difficult to figure out what caused the failure of the overall operation (see figure 1.7).

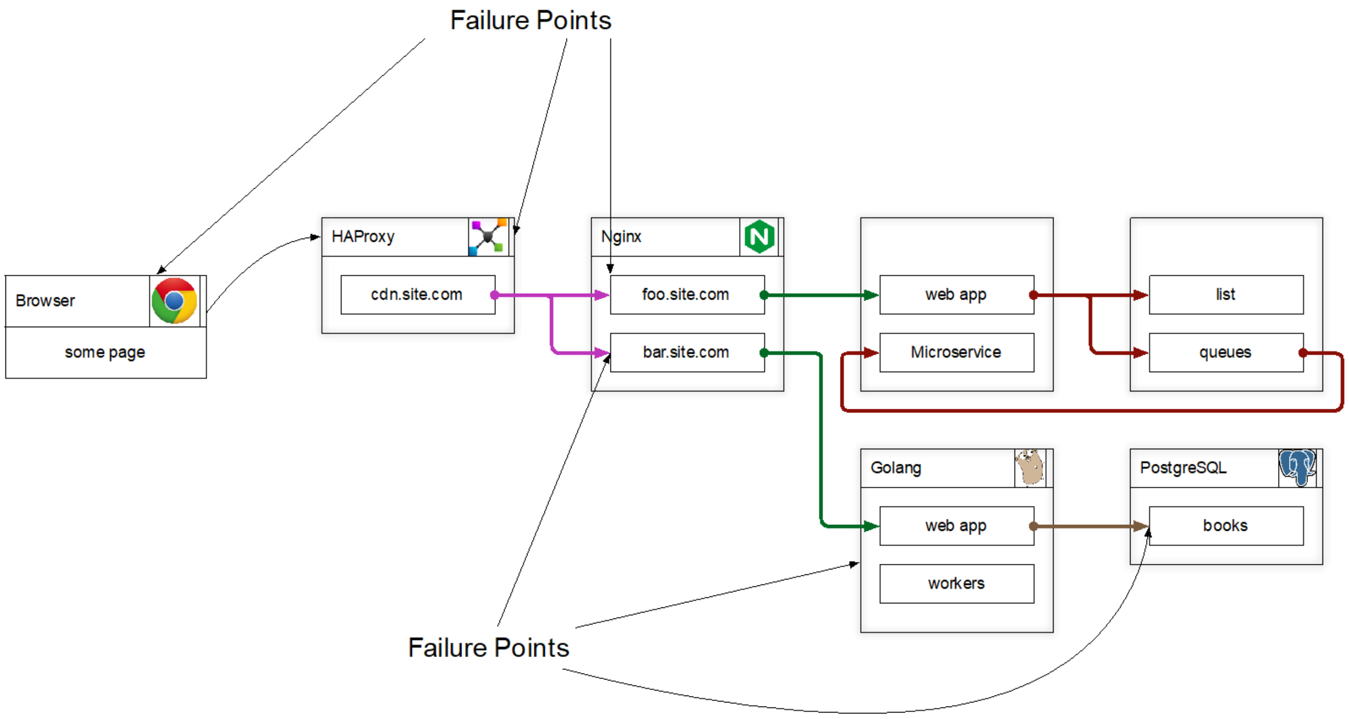


Figure 1.7 You can have many failure points in an integration test. All the units have to work together, and each could malfunction, making it harder to find the source of the bug.

As defined in *The Complete Guide to Software Testing* by Bill Hetzel (Wiley, 1993), integration testing is "an orderly progression of testing in which software and/or hardware elements are combined and tested until the entire system has been integrated." Here's my own variation on defining integration testing.

DEFINITION ***Integration testing*** **is testing a unit of work without having full control over all of its real dependencies, such as other components by other teams, other services, time, network, database, threads, random number generators, and more.**

To summarize: an integration test uses real dependencies; unit tests isolate the unit of work from its dependencies so that they're easily consistent in their results and can easily control and simulate any aspect of the unit's behavior.

1.8.1 Drawbacks of integration tests compared to automated unit tests

Let’s apply the questions from section 1.2 to integration tests and consider what you want to achieve with real-world unit tests:

- *Can I run and get results from the test I wrote two weeks or months or years ago?*

  If you can’t, how would you know whether you broke a feature that you created earlier? Shared data and code changes regularly during the life of an application, and if you can’t (or won’t) run tests for all the previously working features after changing your code, you just might break it without knowing (also known as a ‘regression’). Regressions seem to occur a lot near the end of a sprint or release, when developers are under pressure to fix existing bugs. Sometimes they introduce new bugs inadvertently as they resolve the old ones. Wouldn’t it be great to know that you broke something within *60 seconds* of breaking it? You’ll see how that can be done later in this book.

  DEFINITION **A *regression* is broken functionality. Code that used to work. You can also think of it as one or more units of work that once worked and now don’t.**

- *Can any member of my team run and get results from tests I wrote two months ago?*

  This goes with the previous point but takes it up a notch. You want to make sure that you don’t break someone else’s code when you change something. Many developers fear changing *legacy code* in older systems for fear of not knowing what other code depends on what they’re changing. In essence, they risk changing the system into an unknown state of stability.

  Few things are scarier than not knowing whether the application still works, especially when you didn’t write that code. If you knew you weren’t breaking anything, you’d be much less afraid of taking on code you’re less familiar with, because you have that safety net of unit tests.

  Good tests can be accessed and run by anyone.

  DEFINITION ***Legacy code* is defined by Wikipedia (again, quoting because I don’t necessarily agree) as “source code that relates to a no-longer supported or manufactured operating system or other computer technology,” but many shops refer to any older version of the application currently under maintenance as legacy code. It often refers to code that’s hard to work with, hard to test, and usually even hard to read.**

  A client once defined legacy code in a down-to-earth way: “code that works.” Many people like to define legacy code as “code that has no tests.” *Working Effectively with Legacy Code* by Michael Feathers (Prentice Hall, 2004) uses this as an official definition of legacy code, and it’s a definition to be considered while reading this book.

- *Can I run all the tests I’ve written in no more than a few minutes?*

If you can't run your tests quickly (seconds are better than minutes), you'll run them less often (daily or even weekly or monthly in some places). The problem is that when you change code, you want to get feedback as early as possible to see if you broke something. The more time between running the tests, the more changes you make to the system, and the (many) more places to search for bugs when you find that you broke something.

Good tests should run *quickly*.

- *Can I run all the tests I've written at the push of a button?*

  If you can't, it probably means that you have to configure the machine on which the tests will run so that they run correctly (setting up a docker environment, or setting connection strings to the database, for example) or that your unit tests aren't fully automated. If you can't fully automate your unit tests, you'll probably avoid running them repeatedly, as will everyone else on your team.

  No one likes to get bogged down with configuring details to run tests just to make sure that the system still works. Developers have more important things to do, like writing more features into the system. They can't do that if they don't know the state of the system.

  Good tests should be easily executed in their original form, not manually.

- *Can I write a basic test in no more than a few minutes?*

  One of the easiest ways to spot an integration test is that it takes time to prepare correctly and to implement, not just to execute. It takes time to figure out how to write it because of all the internal and sometimes external dependencies. (A database may be considered an external dependency.) If you're not automating the test, dependencies are less of a problem, but you're losing all the benefits of an automated test. The harder it is to write a test, the less likely you are to write more tests or to focus on anything other than the "big stuff" that you're worried about. One of the strengths of unit tests is that they tend to test every little thing that might break, not only the big stuff. People are often surprised at how many bugs they can find in code they thought was simple and bug free.

  When you concentrate only on the big tests, the overall confidence in your code is still very much lacking. Many parts of the core logic in the code aren't tested (even though you may be covering more components), and there may be many bugs that you haven't considered and might be "unofficially" worried about.

  Good tests against the system should be easy and quick to write, once you've figured out the patterns you want to use to test your specific object model (**REVIEWERS**: I couldn't find a better name for this in the JS space. How would you call the set of design choices and dependencies that led to the current design of your functions, modules and components?).

- Do my tests pass when there are bugs in another team's code? Do my test show the same result when run on different machines or environments? Do my tests stop working if there’s no database, network or deployment?

- These three points refer to the idea that our test code is isolated from various dependencies. The test results are consistent because we have control over what those *indirect inputs* into our system provide. We can have fake databases, fake networks, fate time, fake machine culture. In later chapters I’ll refer to those points as *stubs* and *seams* in which we can inject those stubs.
- If I delete, move or change one test, do other tests remain unaffected?
- Unit tests usually don’t need to have any shared state. Integration tests often do. An external database, external service. Shared state can create a dependency between tests. For example running tests in a the wrong order can corrupt the state for future tests.

Small warning:

Even experienced unit testers can find that it may take 30 minutes or more to figure out how to write the very *first* unit test against an object model they’ve never unit tested before. This is part of the work and is to be expected. The second and subsequent tests on that object model should be very easy to accomplish once you’ve figured out the entry and exit points of the unit of work.

We can recognize three main categories of *things* in the previous questions and answers.

Readability: If we can’t read it, it’s hard to maintain, hard to debug and hard to know what’s wrong.

Maintainability: If it’s painful to maintain the test code or production code because of the tests, our lives will become a living nightmare.

Trust: If we don’t trust our tests when they fail, because we don’t trust the results, we start manually testing again, losing all the time benefit tests are trying to provide. If we don’t trust the tests when they *pass* we start debugging more, again losing any time benefit.

From what I’ve explained so far about what a unit test is not, and what features need to be present for testing to be useful, I can now start to answer the primary question this chapter poses: what’s a good unit test?

1.9 Finalizing our definition

Now that I’ve covered the important properties that a unit test should have, I’ll define unit tests once and for all.

UPDATED AND FINAL DEFINITION 1.2 A *unit test* is an automated piece of code that invokes the unit of work trough an entry point, and then checks one of its exit points. A unit test is almost always written using a unit testing framework. It can be written easily and runs quickly. **It’s trustworthy, readable & maintainable. It is consistent as long as the production code we control has not changed.**

This definition certainly looks like a tall order, particularly considering how many developers implement unit tests poorly. It makes us take a hard look at the way we, as developers, have implemented testing up until now, compared to how we'd like to implement it. (Trustworthy, readable, and maintainable tests are discussed in depth in chapter 8.)

In the first edition of this book, my definition of a unit test was slightly different. I used to define a unit test as "only running against control flow code." But I no longer think that's true. Code without logic is usually used as part of a unit of work. Even properties with no logic will get used by a unit of work, so they don't have to be specifically targeted by tests.

> **DEFINITION** ***Control flow code*** **is any piece of code that has some sort of logic in it, small as it may be. It has one or more of the following: an** `if` **statement, a** `loop`**, , calculations, or any other type of decision-making code.**

Getters/setters are good examples of code that usually doesn't contain any logic and so doesn't require specific targeting by the tests. It's code that will probably get used by the unit of work you're testing, but there's no need to test it directly. But watch out: once you add any logic inside a getter or setter, you'll want to make sure that logic is being tested.

1.10 What About Regression Tests?

Regression testing is defined by Wikipedia (again – due to lack of agreement) as is re-running functional and non-functional tests to ensure that previously developed and tested software still performs after a change.

I'd change that to running *all the tests.*

In this regard, unit tests are just as much regression tests as they are about testing new code. Once we've finished writing them, we should always continue running them for every build, for every commit, to find possible regression issues.

They don't replace higher order functional tests, but they bring a lot of value if done right. Higher order tests are a topic for a future chapter in this book.

In the next section, we'll stop talking about what is a good test, and talk about *when* you might want to write it. I'll discuss test-driven development, because it is often put in the same bucket as doing unit testing. I want to make sure we set the record straight on that.

1.11 Test-driven development

Once you know how to write readable, maintainable, and trustworthy tests with a unit testing framework, the next question is *when* to write the tests. Many people feel that the best time to write unit tests for software is after we've created some functionality and just before we merge our code into remote source control.

Also, to be a bit blunt, a large share don't believe it's a good idea, but have realized through trial and error that there are strict requirements on source control reviews to contain tests, and so they *have* to write tests, in order to appease the code review gods and get their code merged into the main branch (That kind of dynamic of "bad measurement" is a great source of bad tests, and I'll address it in the third part of this book)

A growing number prefer writing unit tests incrementally, during the coding session, and before each piece of very small functionality is implemented. This approach is called test-first or test-driven development (TDD).

NOTE There are many different views on exactly what test-driven development means. Some say its test-first development, and some say it means you have a lot of tests. Some say it's a way of designing, and others feel it could be a way to drive your code's behavior with only some design. For a more complete look at the views people have of TDD, see "The various meanings of TDD" on my blog (http://osherove.com/blog/2007/10/8/the-various-meanings-of-tdd.html). In this book, TDD means test-first development, with design taking an incremental role in the technique (besides this section, TDD isn't discussed in this book).

Figures 1.8 and 1.9 show the differences between traditional coding and TDD.

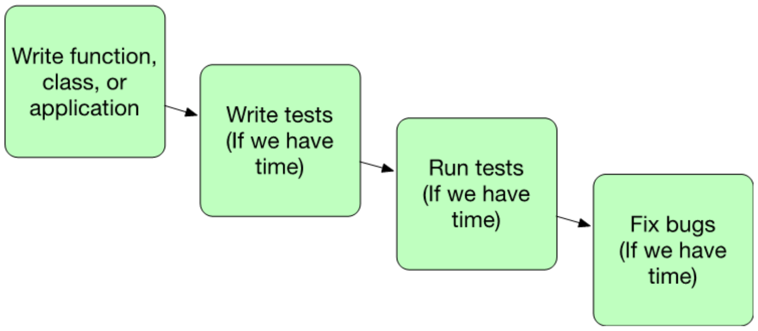

Write function, class, or application
Write tests (If we have time)
Run tests (If we have time)
Fix bugs (If we have time)

Figure 1.8 The traditional way of writing unit tests.

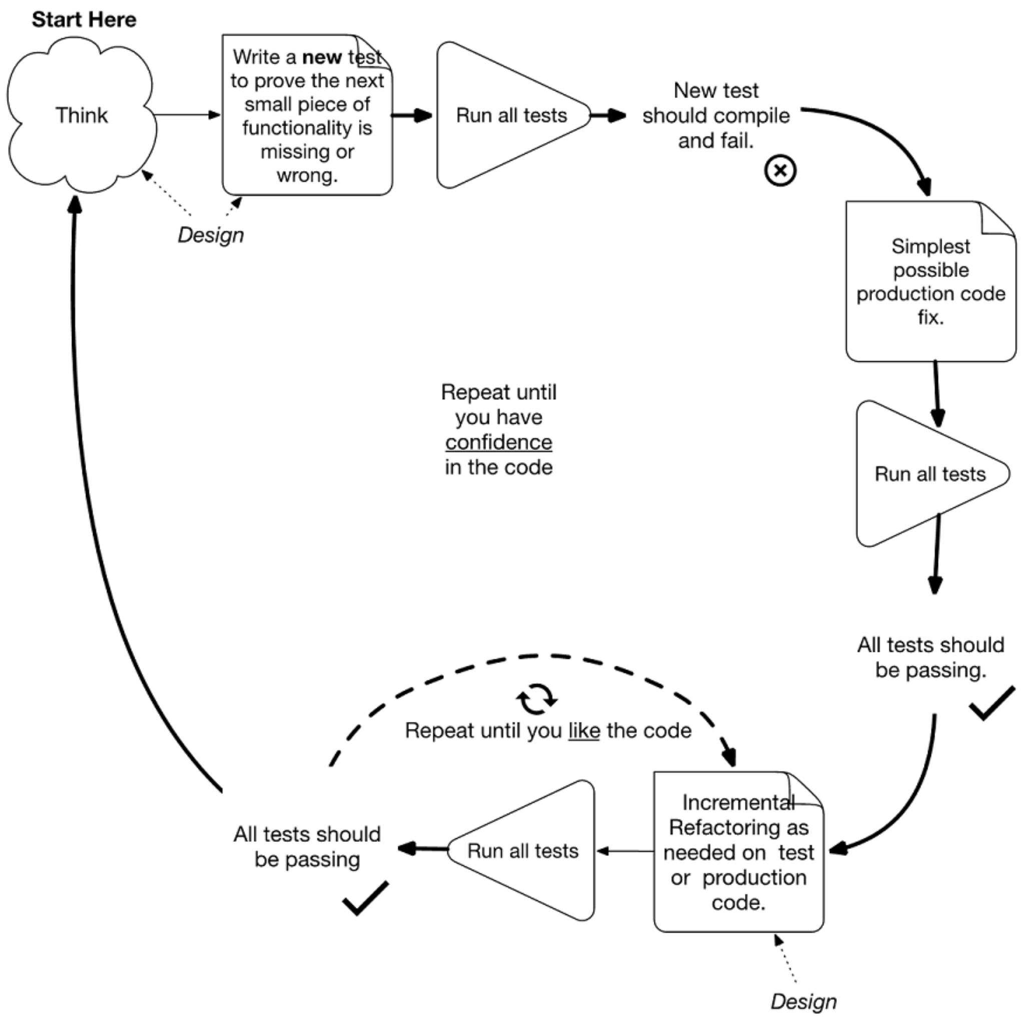

Start Here
Think
Write a **new** test to prove the next small piece of functionality is missing or wrong.
Design
Run all tests
New test should compile and fail.
Simplest possible production code fix.
Run all tests
All tests should be passing.
Repeat until you have confidence in the code
Repeat until you like the code
Incremental Refactoring as needed on test or production code.
Design
Run all tests
All tests should be passing

Figure 1.9 Test-driven development—a bird's-eye view. Notice the circular nature of the process: write test, write code, refactor, write next test. It shows the incremental nature of TDD: small steps lead to a quality end result with confidence.

TDD is different from traditional development, as figure 1.9 shows. You begin by writing a test that fails; then you move on to creating the production code, seeing the test pass, and continuing on to either refactor your code or create another failing test.

This book focuses on the technique of writing good unit tests, rather than on test-driven development, but I'm a big fan of TDD. I've written several major applications and frameworks using TDD, have managed teams that utilize it, and have taught hundreds of courses and workshops on TDD and unit testing techniques. Throughout my career, I've found TDD to be helpful in creating quality code, quality tests, and better designs for the code I was writing. I'm convinced that it can work to your benefit, but it's not without a price (time to learn, time to implement, and more). It's definitely worth the admission price, though, if you're willing to take on the challenge of learning it.

It's important to realize that TDD doesn't ensure project success or tests that are robust or maintainable. It's quite easy to get caught up in the technique of TDD and not pay attention to the way unit tests are written: their naming, how maintainable or readable they are, and whether they test the right things or might have bugs. That's why I'm writing this book – because writing good tests is a separate skill than TDD.

The technique of TDD is quite simple:

1. *Write a failing test to prove code or functionality is missing from the end product.* The test is written *as if* the production code were already working, so the test failing means there's a bug in the production code. How do I know? The test is written such that it would pass if the production code has no bugs.

   In some languages other than JavaScript, since the code doesn't exist yet, the test might not even compile at first. Once it does run, it should be failing since the production code is still not functionally working. This is also where a lot of the "design" in test-driven-design thinking happens.

2. *Make the test pass by adding functionality to the production code that meets the expectations of your test.* The production code should be kept as simple as possible. Don't touch the test. You have to make it pass only by touching production code.
3. *Refactor your code.* When the test passes, you're free to move on to the next unit test or to refactor your code(both production code and the tests) to make it more readable, to remove code duplication, and so on. This is another point where the "design" part happens. We refactor and can even redesign our components while still keeping the old functionality.

Refactoring steps should be very small and incremental, and we run all the tests after each small step to make sure we didn't break anything during out changes. Refactoring can be done after writing several tests or after writing each test. It's an important practice, because it ensures your code gets easier to read and maintain, while still passing all of the previously written tests. We'll have a whole chapter on refactoring later in the book.

DEFINITION ***Refactoring*** **means changing a piece of code *without* changing its functionality. If you've ever renamed a method, you've done refactoring. If you've ever split a large method into multiple smaller method calls, you've refactored your code. The code still does the same thing, but it becomes easier to maintain, read, debug, and change.**

The preceding steps sound technical, but there's a lot of wisdom behind them. Done correctly, TDD can make your code quality soar, decrease the number of bugs, raise your confidence in the code, shorten the time it takes to find bugs, improve your code's design, and keep your manager happier. If TDD is done incorrectly, it can cause your project schedule to slip, waste your time, lower your motivation, and lower your code quality. It's a double-edged sword, and many people find this out the hard way.

Technically, one of the biggest benefits of TDD nobody tells you about is that by seeing a test fail, and then seeing it pass without changing the test, you're basically testing the test itself. If you expect it to fail and it passes, you might have a bug in your test or you're testing the wrong thing. If the test failed and now you expect it to pass, and it still fails, your test could have a bug, or it's expecting the wrong thing to happen.

This book deals with readable, maintainable, and trustworthy tests, but if you add TDD on top, the confidence in your own tests will increase by seeing the tests filing when they should, and passing when they should. In test-after style, you'll usually only see them pass when they should, and fail when they shouldn't (since the code they test should already be working). TDD helps with that a lot, and it's also one of the reasons developers do far less debugging when TDD-ing their code than when they're simply unit testing it after the fact. If they trust the test, they don't feel a need to debug it "just in case." And that's the kind of trust you can only gain by seeing both sides of the test—failing and passing when it should.

1.12 Three core skills needed for successful TDD

To be successful in test-driven development you need three different skill sets: knowing how to write good tests, writing them test-first, and designing the tests and the production code well. Figure 1.10 shows these more clearly.

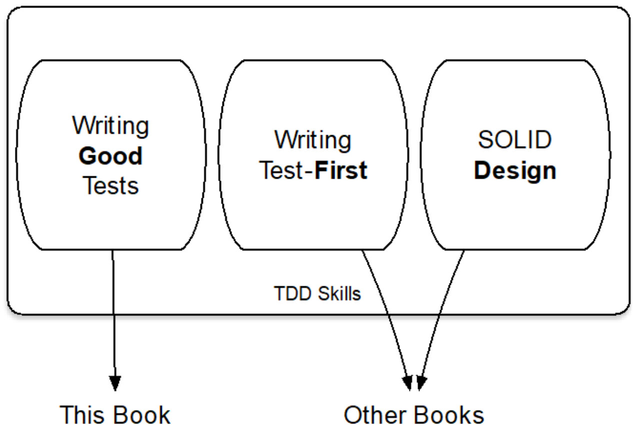


Figure 1.10: Three core skills of Test Driven Development

- *Just because you write your tests first doesn't mean they're maintainable, readable, or trustworthy.* Good unit testing skills are what the book you're currently reading is all about.

- *Just because you write readable, maintainable tests doesn't mean you get the same benefits as when writing them test-first.* Test-first skills are what most of the TDD books out there teach, without teaching the skills of good testing. I would especially recommend Kent Beck's *Test-Driven Development: by Example* (Addison-Wesley Professional, 2002).
- *Just because you write your tests first, and they're readable and maintainable, doesn't mean you'll end up with a well-designed system.* Design skills are what make your code beautiful and maintainable. I recommend *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce (Addison-Wesley Professional, 2009) and *Clean Code* by Robert C. Martin (Prentice Hall, 2008) as good books on the subject.

A pragmatic approach to learning TDD is to learn each of these three aspects separately; that is, to focus on one skill at a time, ignoring the others in the meantime. The reason I recommend this approach is that I often see people trying to learn all three skill sets at the same time, having a really hard time in the process, and finally giving up because the wall is too high to climb.

By taking a more incremental approach to learning this field, you relieve yourself of the constant fear that you're getting it wrong in a different area than you're currently focusing on.

In regard to the order of the learning approach, I don't have a specific scheme in mind. I'd love to hear from you about your experience and recommendations when learning these skills. You can always contact me at at http://contact.osherove.com.

1.13 Summary

In this chapter, we covered the following topics:

- I defined a good unit test as one that has these qualities:
- It should run quickly.
- It should be consistent in its results (it always returns the same result if you don't change anything between runs).
- It should have *full control* of the code under test.
- It should be fully isolated (runs independently of other tests).
- It should run in memory without requiring file system files, networks, databases
- It should be as synchronous and linear as possible.(no parallel threads)
- We also discussed entry points and exit points, exit point types and how they affect our tests.
- We write a very simplistic test harness and API for our production code without using a test framework, to see what it feels like, and how simple the idea is.
- We discussed integration testing vs unit testing.
- Lastly, we looked at test-driven development, how it's different from traditional coding, and what its basic benefits are.

In the next chapter, you’ll start writing your first unit tests using *Jest*, one of the most commonly used test frameworks for JavaScript.

2 A first unit test

This chapter covers

- **Writing your first test with Jest**
- **Test Structure & Naming Conventions**
- **Working with the Assertion AP*is***
- ***Test Refactoring and reducing repetitive code***

When I first started writing unit tests with a real unit testing framework, there was little documentation, and the frameworks I worked with didn't have proper examples. (I was mostly coding in VB 5 and 6 at the time.) It was a challenge learning to work with them, and I started out writing rather poor tests. Fortunately, times have changed. In JavaScript, and in practically any language out there, there's a wide range of choices and plenty of documentation and support from the community to try out these bundles of helpfulness.

in the previous chapter we wrote a very simplistic home-grown test framework. In this chapter we'll take a look at *Jest* as our framework of choice for this book.

2.1 About Jest

Jest is an open source test framework created by *Facebook*. It was originally created for testing frontend *React* components in JavaScript. These days it's widely used in many parts of the industry for both backend and frontend projects testing. It supports two major flavors of test syntax (one that uses the word 'test' and and another that is based on the 'Jasmin'

Syntax – a framework that has inspired many of Jet's features). We'll try both of them to see which one we like better.

Jest is easy to use, easy to remember, and has lots of great features.

Aside from Jest, there are many other frameworks in JavaScript, pretty much all open source as well. There are some differences between them in style and Api*s,* but for the purposes of this book, it shouldn't matter too much. For reference, I'll cover them in *the appendix* relating *to tools at the end of the book.*

2.1.1 Preparing our environment

- Make sure you have node.js installed locally. You can follow the instructions over at https://nodejs.org/en/download/ to get it up and running on your machine. The site will provide with an option of either an 'LTS' (long erm support' release or a 'current' release. The LTS release is geared towards enterprises and the current release has a more frequent update nature. Either will work for the purposes of this book.
- Make sure that the Node Package Manager (npm for short) is installed on your machine. It is included with node.js so run the command npm -v on the command line and if you see an output similar to 6.10.2 or higher you should be good to go. If not, make sure node.js installed (see previous point)

2.1.2 Preparing our working folder

- To get started with *Jest* let's create a new empty folder named `ch2` and initialize it for our purposes with a package manager of our choice. Jest expects either a `jest.config.js` or a `package.json` file. We're going with the latter – `npm init` will generate one for us.

  ```
  mkdir ch2
  cd ch2
  npm init --yes
  //or
  yarn init –yes
  git init
  ```

  Yarn is an alternative package manager. It shouldn't matter for the purposes of this book which one you use. I'll just use npm since I have to choose one.

  I'm also initializing git in this folder. This would be a recommended idea anyway to track changes, but for Jest specifically this file is used under the covers to track changes to files and run specific tests. It makes Jest's life easier.

  By default, jest will look for its configuration either at the package.json file that is created by this command or in a special jest.config.js file. For now, we won't need anything but the default package.json file. If you'd like to learn more about all the jest configuration options, refer to https://jestjs.io/docs/en/configuration

2.1.3 Installing Jest

Next, we'll install Jest. To install *jest* as a *dev dependency* (which means it does not get distributed to production) we can write the command:

```
npm install --save-dev jest
//or
yarn add jest –dev
```

This will end up with a new file jest.js under our `[root folder]/node_modules/bin`. We can then execute Jest using the command `npx jest`.

We can also install Jest *globally* on the local machine (I recommend doing this on top of the save-dev installation) by executing:

```
npm install -g jest
```

This would give us the freedom later on to just execute the command 'jest' in any folder that has tests directly in the command line without going through `npm` to execute it.

In real projects, it is common to use `npm` commands to run tests instead of using the global jest. I'll show how this is done in the next few pages.

2.1.4 Creating a test file

Jest has a couple of default ways to find test files.

- If there's a __tests__ folder, it loads all the files in it as test files, regardless of their naming conventions.
- Any file that ends with *.spec.js or *.test.js , in any folder under the root folder of your project, recursively.

We'll use the first variation, but also name our files with either *test.js or *.spec.js to make things a bit more consistent if we want to move them around later on (and stop using __tests_ altogether)

You can also configure jest to your heart's content on how to find which files where under a jest.config.js file or through package.json. You can look up the jest docs over at https://jestjs.io/docs/en/configuration to find out all the gory details.

- The next step is to create a special folder under our `ch2` folder called `__tests__`
- Under this folder we can create a file that ends with either `test.js` or `spec.js`. for example, `my-component.test.js`

  Which suffix you choose it up to you. Just be consistent. It's about your own style. I'll use them interchangeably in this book because I think of 'test' as the most simplistic version of 'spec' – so I use it when showing very simple things.

Test File Locations

There are two main patterns I see for placing test files. Some people prefer to place the test files directly next to the files or modules being tested. Others prefer to place all the files under a test directory. It doesn't really matter much, just be consistent in your choice throughout the project, so it's easy to know where to find the tests for a specific item.

I find that placing tests in a test folder allows me to also put helper files under the tests folder close to the tests. As for easily navigating between tests and code under tests, there are plugins for most IDEs today that allow with a keyboard shortcut to navigate between code and its tests and vice versa.

We don't need to `require()` at the top of the file.any code to start using jest. It automatically imports global functions for us to use. The main functions you should be interested in include `test`, `describe`, `it` and `expect` . Here's how a simple test might look:

Listing 2.1: Hello Jest

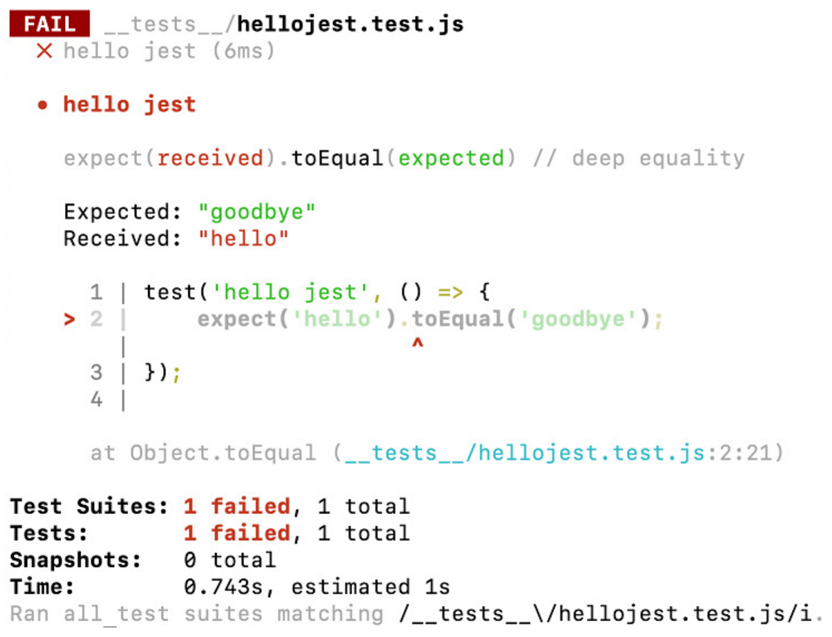
```
// hellojest.test.js

test('hello jest', () => {
    expect('hello').toEqual('goodbye');
});
```

We haven't used `describe` and `it` yet, but we'll get to them soon.

2.1.5 Executing Jest:

To run this test we can need to be able to execute jest. To recognize it from the command line we'd need to do either of the following:

1. Install Jest globally on the machine by running npm install jest -g
2. Use npx to execute jest from the node_modules directory by typing the following:
3. the command line `‘jest’` in the root of the folder `ch2`. If all the stars lined correctly up we should see the results of the jest test run and a failure. Our first failure. Yay! Figure 2.1 shows the output on my terminal when I run the command:

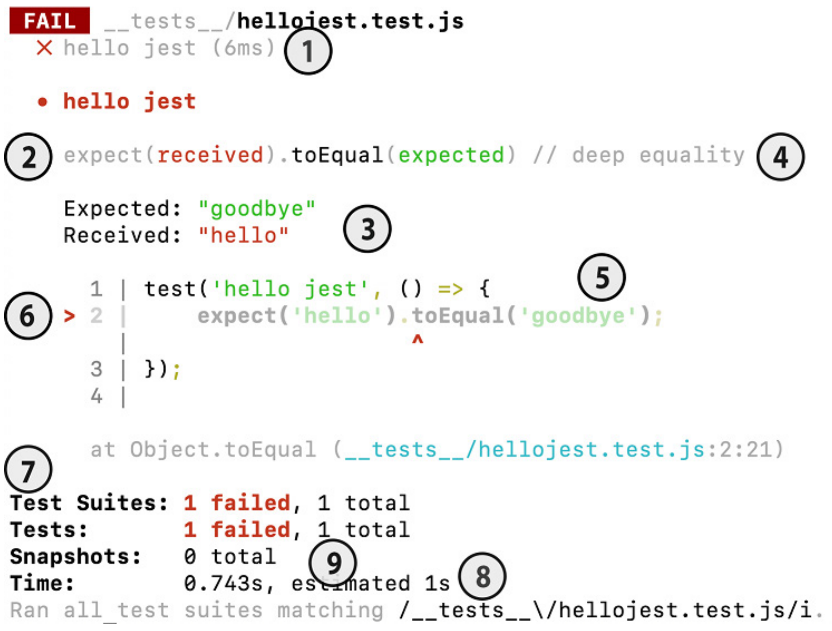


```
 FAIL  __tests__/hellojest.test.js
  ✕ hello jest (6ms)

  ● hello jest

    expect(received).toEqual(expected) // deep equality

    Expected: "goodbye"
    Received: "hello"

      1 | test('hello jest', () => {
    > 2 |     expect('hello').toEqual('goodbye');
        |                     ^
      3 | });
      4 |

      at Object.toEqual (__tests__/hellojest.test.js:2:21)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.743s, estimated 1s
Ran all test suites matching /__tests__\/hellojest.test.js/i.
```

Figure 2.1 Terminal Output from Jest

It’s pretty cool to see such a lovely, colorful (if you’re reading the e-book version of this), useful output from a test tool. It looks even cooler if you have a dark screen in your terminal.

Let’s take a closer look at each part. Figure 2.2 is the same but with numbers to follow along:

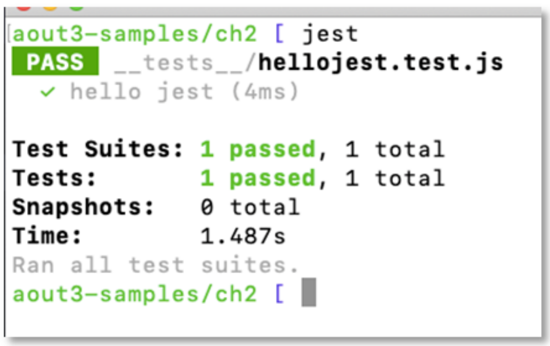


```
FAIL  __tests__/hellojest.test.js
  ✕ hello jest (6ms)

  ● hello jest

    expect(received).toEqual(expected) // deep equality

    Expected: "goodbye"
    Received: "hello"

      1 | test('hello jest', () => {
    > 2 |     expect('hello').toEqual('goodbye');
        |                     ^
      3 | });
      4 |

      at Object.toEqual (__tests__/hellojest.test.js:2:21)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.743s, estimated 1s
Ran all test suites matching /__tests__\/hellojest.test.js/i.
```

Figure 2.2 Terminal Output from Jest

Let’s see how many pieces of information are presented here:

1. A quick list of all the failing tests (with names) with nice red xs next to them.
2. Detailed report on the expectation that failed (a.k.a our assertion).
3. The exact difference between actual value and expected value
4. The type of comparison that was executed
5. The code for the test
6. Pinpointing the exact line (visually) where the test failed.
7. Report of how many tests run, failed and passed
8. Time it took
9. Number of snapshots (not relevant to our discussion)



```
aout3-samples/ch2 [ jest
 PASS  __tests__/hellojest.test.js
  ✓ hello jest (4ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.487s
Ran all test suites.
aout3-samples/ch2 [ 
```

Imagine trying to write all this nice reporting functionality yourself. It's possible, but who's got the time, and the inclination, plus take care of any bugs in the reporting mechanism.

If we fix '`hello`' to '`goodbye`' in the test, we can see what happens when the test passes:

Nice and green, as all things should be (again – in the digital version otherwise it's nice and grey).

You might note that it takes 1.5 seconds to run a single hello world kind of test. If we used the command `jest --watch` instead, we can have Jest monitor filesystem activity in our folder and automatically run tests for files that have changed without re-initializing itself every time. This can save a considerable amount of time and really helps with the whole notion of 'continuous testing'.

Set a terminal in the other window of your workstation with `jest --watch` on it, and keep coding and getting fast feedback on issues you might be creating. That's a good way to get into the flow of things.

Jest also supports async style testing and callbacks. I'll touch on these as needed as we get to those topics later on in the book, but for now if you'd like to learn more about this style head over to the *Jest* documentation on the subject: https://jestjs.io/docs/en/asynchronous .

2.2 The Library, the Assert, the Runner & the Reporter

Jest acted in several capacities for us:

- It acted as a **test library** to use when writing the test
- It acted as an **assertion library** for **asserting** inside the test ('`expect`')
- It acted as the **test runner**
- It acted as the **test reporter** for the test run.

We also haven't seen this yet, but it also provides *isolation* facilities to create mocks, stubs and spies. We'll touch on these ideas in later chapters.

Other than isolation facilities, It is very common in other languages that the test framework fills all the roles I just mentioned: Library, assertions, test runner and test reported. But the JS world seems a bit more fragmented. Many other test frameworks provide only some of these facilities. Perhaps this is because the mantra of "do one thing, and do it well" has been taken well to heart, and perhaps it's other reasons. I'm not really sure why. *Jest* stands out as one of a handful of all-in-one frameworks. It is a testament to the strength of the open source culture in javascript to see that for each one of these categories there are multiple tools that you can mix and match to create your own super-toolset. I touch on the various tools in each category in the tools and frameworks appendix. One of the reasons I chose *Jest* for this book is so we don't have to bother ourselves too much with the tooling aspects or missing features, and just worry about the patterns. That way we won't have to use multiple frameworks in a book that is mostly concerned with patterns and anti patterns.

2.3 What unit testing frameworks offer

My editor believes this section belongs at the beginning of this chapter, but I wanted to get you to jump in and get your hands dirty as soon as possible, so that when we get to this section, it will be based on some experience trying to write a couple of small tests.

So, with that out of the way, let's zoom out for a second and see where we are.

What do frameworks like jest offer us over creating our own framework like we started doing in the previous chapter or over manually testing things?

- *Structure. Instead of* reinventing the wheel every time you want to test a feature, when using a test framework, you always start out the same way – by writing a test with a well-defined structure that everyone can easily recognize, read and understand.
- *Repeatability.* When using a test framework, it's easy to repeat the act of writing a new test. it's also easy to repeat the execution of the test, using a test runner and it's easy to do this fast and many times a day. It's also easy to understand failures and their reasons why, instead of us coding all that stuff into our hand-rolled framework. Someone already did all the hard work for us.
- *Confidence through time savings.*

  When we roll our own test framework, we are more likely to have more bugs in it, since it is less battle tested than an existing mature and widely used framework.

  On the other hand, if we're just manually testing *things* it's usually very time consuming. When we're short on time, we'll only focus on testing the things that feel the most "severe" and skip over things that might feel less important to test. We could be skipping small but significant bugs. *by making it easy to write new tests it's more likely that we'll also write tests for the stuff that feels "smaller" because we won't spend too much time writing tests for the big stuff.*

  One extra benefit might be that the reporting can be helpful to manage tasks at the team level (test is passing === task is done). Some people find this useful.

In short, *frameworks* for writing, running, and reviewing unit tests and their results can make a huge difference in the daily lives of developers willing to invest the time to learn how to use them properly. Figure 2.2 shows the areas in software development where a unit testing framework & it's helper tools have influence.

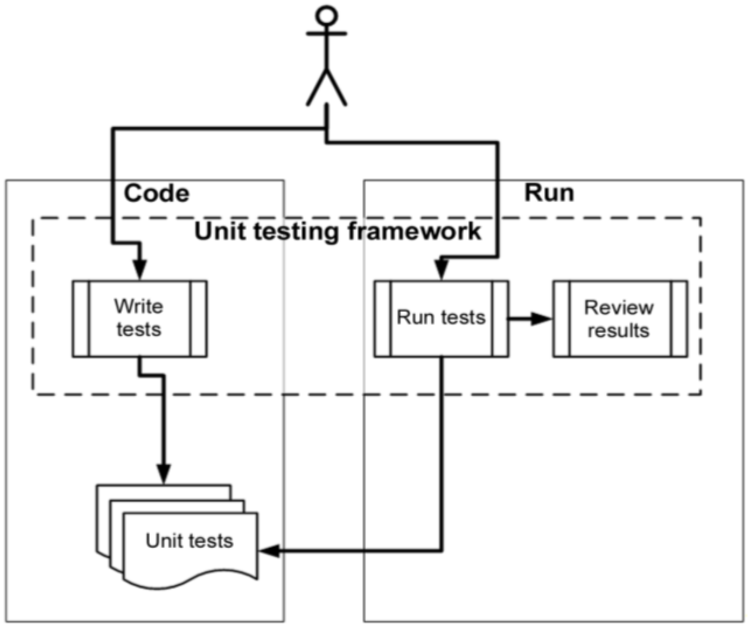


Figure 2.2 Unit tests are written as code, using libraries from the unit testing framework. Then the tests are run from a test runner inside the IDE or through command line, and the results are reviewed through a test reporter (either as output text, the IDE) by the developer or an automated build process.

Here are the types of actions we usually execute with a test framework:

Table 2.1 How testing frameworks help developers write and execute tests and review results

Unit testing practice	How the framework helps
Write tests easily and in a structured manner.	Framework supplies the developer with helper functions, assertion functions and structure related functions.
Execute one or all of the unit tests.	A test runner (command line usually) that • Identifies tests in your code • Runs tests automatically • Indicates status while running
Review the results of the test runs.	The test runners will usually provide information such as • How many tests ran • How many tests didn't run • How many tests failed • Which tests failed

- The reason tests failed
- The code location that failed
- Possibly a full stack trace of any exceptions that caused the test to fail, and will let you go to the various method calls inside the call stack

At the time of this writing, there are around 900 unit testing frameworks out there—with more than one a couple for most programming languages in public use (and a few dead ones). You can find a good list at https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks .

NOTE Using a unit testing framework doesn't ensure that the tests you write are *readable*, *maintainable*, or *trustworthy* or that they cover all the logic you'd like to test. We'll look at how to ensure that your unit tests have these properties in chapter 7 and in various other places throughout this book.

2.3.1 The xUnit frameworks

When I started writing tests (this was still in the visual basic days) the standard by which most unit test frameworks were measured was collectively called "XUnit". The grandfather of the "XUnit" frameworks idea was *SUnit*, the unit test framework for SmallTalk.

These unit testing frameworks' names usually start with the first letters of the language for which they were built; You might have CppUnit for C++, JUnit for Java, NUnit & XUnit for .NET, and HUnit for the Haskell programming language. Not all of them follow these naming guidelines, but most do.

2.3.2 XUnit, TAP and Jest Structures

It's not just the name, though. If you were using "an XUnit framework", you could also expect a specific structure in which the tests are built. This was usually

- "Test Suite" that contains
- "Fixtures" that contain
- "Test Cases".

And when these frameworks would run, they would output their results in the same structure as well, which was usually an XML file with a specific schema.

This type of XUnit XML report is still prevalent today – and is widely used in most build tools such as Jenkins support this format with native plugins and use it to report the results of test runs.

Today, most unit test frameworks in static languages still use the XUnit model for structure, which means, once you've learned to use one of them, you should be able to easily use any of them (assuming you know the particular programming language.

The other interesting standar for the reporting structure of test results and more, is called "`TAP`" – "Test Anything Protocol". `TAP` started life as part of the test harness for Perl but now has implementations in C, C++, Python, PHP, Perl, Java, JavaScript, and others. TAP is much more than just a reporting specification. In the JS world, the *tape* framework is the most well known test framework to natively support the TAP protocol.

Jest is not strictly an XUnit or TAP framework. Its output is not XUnit or TAP compliant by default. Because XUnit style reporting still rules the build-sphere, we 'll usually want to adapat to that protocol for our reporting on a build server. To get jest test results to be easily recognized by most build tools, you can install npm modules such as *jest-xunit* (if you want TAP specific output, use *jest-tap-reporter)* and then use a special *jest.config.js* file in your project to configure jest to alter it reporting format.

Moving on. Let's write something that feels a bit more like a real test with Jest, shall we?

2.4 Introducing the Password Verifier Project

The project that we'll mostly use for testing examples in this book will be simple at first and will contain only one function. As the book moves along, we'll extend that project with new features, modules and classes to demonstrate certain aspects. We'll call it the `Password Verifier` project.

The first scenario is pretty simple. We are building a password verification library. It will be just a function at first. The function, `verifyPassword(rules)`, allows putting in custom verification functions dubbed `rules`, and outputs the list of errors per the rules that have been input into the function. Each rule function will output two fields:

```
{
 passed: (boolean),
      reason: (string)
 }
```

In this book, I'll teach you to write tests that check `verifyPassword`'s functionality in multiple ways as we add more features to it.

Listing 2.2 shows version 1.0 of this function, with a very naïve implementation:

Listing 2.2: passwordVerifier version 0

```
// ch2/password-verifier0.js

const verifyPassword = (input, rules) => {
  const errors = [];
  rules.forEach(rule => {
    const result = rule(input);
    if (!result.passed) {
      errors.push(`error ${result.reason}`);
    }
  });
  return errors;
};
module.exports = { verifyPassword };
```

Granted, it's not the most "functional" styled code, and we might refactor it a bit later, but I wanted to keep things very simple at the beginning so we can focus on the tests.

The function doesn’t really do much. It iterates over all the rules given, and runs each one with the supplied input. If the rule’s result is not *passed* then an error is added to the final errors array which is returned as the final result.

2.5 The first Jest test for verifyPassword

Assuming we have Jest installed, we can go ahead and create a new file under the `__tests__` folder named `password.verifier.spec.js`.

The __tests_ folder is only one convention on where to organize and place your tests, and it’s part of jest’s default configuration. There are many that prefer placing the test files right alongside the code being tested. There are pros and cons for each way but we’ll get into that in later parts of the book. For now, we’ll go with the defaults.

Here’s a first version of a test against our new function:

Listing 2.3 – the first test against passwordVerifier0

```
// password.verifier0.spec.js

1: const { verifyPassword } = require('../password-verifier0');
2:
3: test('badly named test', () => {
4:   const fakeRule = input => ({ passed: false,
5:                                reason: ‘fake reason’ });
6:   const errors = verifyPassword('any value', [fakeRule]);
7:   expect(errors[0]).toMatch('fake reason');
8: });
```

Unit tests usually are made of three main “steps”: Arrange, Act and Assert

1. Arrange: setup the scenario under test. (line 4)
2. Act: invoke the entry point with inputs (line 6)
3. Assert: check the exit point. (line 7)

This is colloquially called the ‘AAA’ Pattern. It’s quite nice! I find it very easy to reason about the “parts” of a test by saying things like “that arrange part is too complicated” or “where is the “Act” part?” .

On line 4, We’re creating a fake rule that always returns false, so that we can prove it’s actually used by asserting *on its reason* at the end of the test on line 7. We then send it to our `verifyPassword` along with a simple input. We then assert on line 7 that the first error we get matches the fake reason we gave in line 5. `.toMatch(/string/)` uses a regular expression to find a part of the string. It’s the same as using `.toContain(‘fake reason’)`.

It’s getting tedious to run jest manually after we write a test or fix something, so let’s configure `npm` to run jest automatically.

Go to `package.json` in the root folder of ch2 and provide the following items under the *scripts* item:

```
"scripts": {
    "test": "jest",
    "testw": "jest --watch" //if not using git, change to --watchAll
  },
```

If we don’t have git initialized in this folder, we can use the command `--watchAll` instead of `--watch`.

If everything went well we can now type `npm test` in the command line from `ch2` folder, and jest will run the tests once. If we type `npm run testw`, jest will run and wait for changes, in an endless loop until we kill the process with ctrl-c. (we need to use the word ‘`run`’ since `testw` is not one of the special npm keywords that npm recognizes automatically).

If we run the test we can see that it passes, since the function works as expected.

2.5.1 Testing the test

Let’s put a bug in the production code and see that the test fails when it should:

Listing 2.4 – adding a bug

```
const verifyPassword = (input, rules) => {
  const errors = [];
  rules.forEach(rule => {
    const result = rule(input);
    if (!result.passed) {
      // we've accidentally commented out the line below
      // errors.push(`error ${result.reason}`);
    }
  });
  return errors;
};
module.exports = { verifyPassword };
```

We should now see our test failing with a nice message. Let’s uncomment the line and see the test pass again. This is a great way to gain some confidence in your tests if you’re not doing test-driven-development, and are writing the tests after the code.

2.5.2 U.S.E Naming

Our test has a really bad name. It doesn’t explain anything about what we’re trying to accomplish here. I like to put three pieces of information in test names, so that the reader of the test will be able to answer most of their mental questions just by looking at the name of the test. These three parts include:

1. The unit of work under test (the `verifyPassword` function in this case)
2. The scenario or inputs to the unit (failed rule)
3. The expected behavior or exit point (returns an error with a reason)

During the review process, Tyler Lemke, a reviewer of the book had come up with a nice acronym for this: U.S.E: Unit under test, Scenario, Expectation. I like it and it’s easy to remember. Thanks Tyler!

Listing 2.5 shows our next revision of the test with a U.S.E name:

Listing 2.5 – naming a test with U.S.E

```
1: test('verifyPassword, given a failing rule, returns errors', () => {
2:   const fakeRule = input => ({ passed: false, reason: 'fake reason' });
3:
4:   const errors = verifyPassword('any value', [fakeRule]);
5:   expect(errors[0]).toContain('fake reason');
6: });
```

This is a bit better. When a test fails, especially during a build process, you don't see comments, you don't see the full test code. You usually only see the name of the test. The name should be so clear that you might not even have to look at the test code to understand where the production code problem might be.

2.5.3 String comparisons and maintainability

We also made another small change in line 5:

```
5:   expect(errors[0]).toContain('fake reason');
```

Instead of checking that one string is equal to another as is very common in tests, we are checking that a string is contained in the output. This makes our test less brittle for future changes to the output. We can use `toContain` or `.toMatch(/fake reason/)` – which uses a regular expression to match a part of the string, to achieve this.

Strings are a form of user interface. They are visible by humans and they might change. Especially the edges of strings. We might add whitespace, tabs, asterisks and other embellishments to a string.

We care that the *core* of the information contained in the string exists. We don't want to go and change our test every time someone adds a new line to the ending of it. This is part of the thinking we want to encourage in our tests: test maintainability over time, and its brittleness – are of high priority.

We'd like the test to ideally only fail when something actually is wrong in the production code. We'd like to reduce the number of false positives to a minimum. `toContain()` or `toMatch()` are great ways to move towards that goal.

I talk about more ways to improve test maintainability throughout the book, and especially in the second part of the book.

2.5.4 Using describe()

We can use Jest's `describe` function to create a bit more structure around our test, and to start separating the three pieces of information from each other. This step, and the ones after it are completely up to how you want to style your test and its readability structure. I'm showing them here because many people either don't use describe in an effective way, or they ignore it all together. It can be quite useful.

`describe` functions wrap our tests with context: both logical for the reader, and functional for the test itself. Here's how we can start using them:

Listing 2.6 – adding a describe() block

```
// ch2/__tests__/password.verifier0.spec.js
01: describe('verifyPassword', () => {
02:   test('given a failing rule, returns errors', () => {
03:     const fakeRule = input =>
04:         ({ passed: false, reason: 'fake reason' });
05:
06:     const errors = verifyPassword('any value', [fakeRule]);
07:
08:     expect(errors[0]).toContain('fake reason');
09:   });
10: });
```

I've made three changes here:

1. I've added a `describe` block that describes the unit of work under test.

   To me this looks clearer. It also feels like I can now add more nested tests under that block. This also helps the command line reported create nicer reports.

2. I've nested the `test` under the new block and removed the name of the unit of work from the test.
3. I've added the `input` into the fake rule's `reason` string in line 4.

   Important: This 3rd change is not actually a great idea in terms of making the test's code more complex than it needs to be. Also it might lead to a potential bug in our tests: String interpolation is a form of logic. I try to avoid logic in my unit tests like the plague. No ifs, switches, loops or other forms of logic if I can help it, and in unit tests, I usually can. The bug risk is too high and finding bugs in tests can be very bad for morale, and very good for alcohol vendors. This may seem like an overreaction over such a small thing, but it's a very slippery slope, so I just avoid the slope in the first place if I can help it.

   I'll talk more about test logic in the second part of the book.

4. I've also added an empty line between the arrange, act and assert parts to make the test more readable, especially to someone new to the team.

2.5.5 Structure can imply context

The nice thing about `describe` is that is can be nested under itself. So we can use it to create another level that explains the scenario, and under that we'll nest our test. Like so:

Listing 2.7 – Nested describes for extra context

```
// ch2/__tests__/password.verifier0.spec.js

describe('verifyPassword', () => {
  describe('with a failing rule', () => {
    test('returns errors', () => {
      const fakeRule = input => ({ passed: false,
                                   reason: 'fake reason' });

      const errors = verifyPassword('any value', [fakeRule]);

      expect(errors[0]).toContain('fake reason');
    });
  });
});
```

Some people will hate it. I think there is a certain elegance to it. This nesting allows us to separate the three pieces of critical information to each level here. In fact, we can also extract the false rule outside of the test right under the relevant describe if we wish to:

Listing 2.8 – Nested describes for extra context

```
// ch2/__tests__/password.verifier0.spec.js

describe('v1.4: verifyPassword', () => {
  describe('with a failing rule', () => {
    const fakeRule = input => ({ passed: false,
                                  reason: 'fake reason'
    });
    test('returns errors', () => {
      const errors = verifyPassword('any value', [fakeRule]);

      expect(errors[0]).toContain('fake reason');
    });
  });
});

```

For the next example I'll move this rule back into the test (I like it when things are close together – more on that later

This nesting structure also implies very nicely that under a specific scenario you could have more than one expected behavior. You could check multiple *exit points* under a scenario, each one as a separate test, and it still makes sense from the reader's point of view.

2.5.6 The 'It' function

There's one missing piece to the puzzle I'm building so far. Jest also exposes an `it` function. This function is for all intents and purposes, an *alias* to the `test`*()* function, but it fits in more nicely in terms of syntax with the describe-driven approach outlined so far.

Here's how the test looks when I replace *test with it.*

Listing 2.9 – replacing test() with it()

```
// ch2/__tests__/password.verifier0.spec.js


describe('verifyPassword', () => {
  describe('with a failing rule', () => {
    it('returns errors', () => {
      const fakeRule = input => ({ passed: false,
                                  reason: 'fake reason' });

      const errors = verifyPassword('any value', [fakeRule]);

      expect(errors[0]).toContain('fake reason');
    });
  });
});
```

In this test, it is very easy to understand what `‘it’` refers to. It is a natural extension of the `describe` blocks above it. Again, it’s up to you whether you want to use this style. I’m showing one variation of how I like to think about it.

2.5.7 Two Jest Flavors

As we’ve seen, Jest supports two main ways to write tests. A terse `‘test’` syntax, and a more `describe`-driven (i.e hierarchical) syntax. The `describe`-driven inspiration of the Jest syntax can be largely attributed to `Jasmine`, One of the oldest JS test frameworks. The style itself can be traced back to Ruby land and the well known `RSpec` Ruby test framework.

This nested style is usually called “BDD Style” for “Behavior driven development”.

You can mix and match them as you like (I do). You can use the ‘test’ syntax when it’s easy to understand your test target and all of its context without going into too much trouble. The describe syntax can help when you’re expecting multiple results from the same entry point under the same scenari I’m showing them both here because I sometimes use the terse flavor, and sometimes use the `describe`-driven flavor, depending on the complexity and expressiveness require

BDD’s Dark Present

BDD itself has quite an interesting background that might be worth talking about a bit BDD isn’t related to TDD. Dan North, the person most associated with inventing this term, refers to it as using stories and examples to describe how an application should behave. Mainly this is targeted at working with non-technical stakeholders. Product owners, customers etc.

`Rspec` **(inspired by** `RBehave`**) brought the story driven approach to the masses and in the process, many other frameworks had come along, including the famous** `cucumber`**.**

There is also a dark side to this story: many frameworks have been developed and used solely by developers without working with non-technical stakeholders, in complete opposition to the main ideas of BDD.

Today, the term ‘BDD frameworks” to me mainly means “test frameworks with some syntactic sugar”, since they are almost never used to create real conversations between stakeholders, and almost always as just another shiny tool or a prescribed tool to perform developer based automated tests.

I’ve even seen the mighty `cucumber` **fall into this pattern.**

2.5.8 Refactoring the Production Code

Since there are many ways to build the same thing in JavaScript, I thought I’d show a couple of detours in our design, and what happens if we change it.

Say we’d like to make the `password verifier` an object with state?

One reason to change the design into a stateful one might be that I intend for different parts of the application to use this object. One part will configure and add rules to it, and a different part will use it to do the verification.

Another justification is that we need to know how to handle a stateful design and look at which directions it pulls our tests, and what we can do about those directions.

Let’s see the production code first:

Listing 2.10 – refactoring a function to a stateful class

```
//ch2/password-verifier1.js

class PasswordVerifier1 {
  constructor () {
    this.rules = [];
  }

  addRule (rule) {
    this.rules.push(rule);
  }

  verify (input) {
    const errors = [];
    this.rules.forEach(rule => {
      const result = rule(input);
      if (result.passed === false) {
        errors.push(result.reason);
      }
    });
    return errors;
  }
}
module.exports = { PasswordVerifier1 };
```

I’ve highlighted the main changes in listing 2.9. There’s nothing really special going on here. This should feel more comfortable if you’re coming from an object oriented background. It’s important to note that this is just one way to design this functionality, and I’m using the class based approach so that I can show how this design affect our test.

In this new design, where is our entry point and exit point for the current scenario? Think about it for a second. The scope of the unit of work has increased. To test a scenario with a failing `rule` we would have to invoke two functions that affect the state of the unit under test: `addRule` and `verifyPassword`.

Now let’s see how the test might look (changes are highlighted, as usual):

Listing 2.11 – Testing the stateful unit of work

```
//ch2/__tests__/password.verifier1.spec.js

const { PasswordVerifier1 } = require('../password-verifier1');

describe('PasswordVerifier', () => {
  describe('with a failing rule', () => {
    it('has an error message based on the rule.reason', () => {
      const verifier = new PasswordVerifier1();
      const fakeRule = input => ({passed: false,
                                    reason: 'fake reason'});

      verifier.addRule(fakeRule);
      const errors = verifier.verify('any value');

      expect(errors[0]).toContain('fake reason');
    });
  });
});
```

So far, so good. nothing fancy is happening here. Note that the surface of the unit of work has increased. It now spans two functions that are related and must work together (`addRule` and `verify`). There is a *coupling* that occurs due to the stateful nature of the design. We need to use two functions to test productively without exposing any internal state from the object.

The test itself looks innocent enough. What happens when we want to write several tests for the same scenario? That would happen when we have multiple *exit points*, or if we want to test multiple results from the same *exit point*. For example, Let's say we want to verify we only have a *single error*. We could simply add a line to the test like this:

```
1: ...
2: verifier.addRule(fakeRule);
3:       const errors = verifier.verify('any value');
4:       expect(errors.length).toBe(1);
5:       expect(errors[0]).toContain('fake reason');
6: ...
```

What happens if line 4 fails? Line 5 never executes, because the test runner will receive an error, and move on to the next test case.

We'd still want to know if line 5 would have passed, right? So maybe we'd start commenting out line 4 and re-running the test. That's not a healthy way to run your tests. In Gerard Meszaros' book "XUnit Test Patterns" this human behavior of commenting things out to test other things is called "Assertion Roulette". It can create lots of confusion and false positives in your test runs (thinking that something is failing or passing when it isn't).

I'd rather separate this extra check into its own test case with a good name like so:

Listing 2.12 – Checking an extra end result from the same exit point

```
//ch2/__tests__/password.verifier1.spec.js
const { PasswordVerifier1 } = require('../password-verifier1');
describe('v3 PasswordVerifier', () => {
  describe('with a failing rule', () => {
    it('has an error message based on the rule.reason', () => {
      const verifier = new PasswordVerifier1();
      const fakeRule = input => ({passed: false,
                                  reason: 'fake reason'});

      verifier.addRule(fakeRule);
      const errors = verifier.verify('any value');

      expect(errors[0]).toContain('fake reason');
    });
    it('has exactly one error', () => {
      const verifier = new PasswordVerifier1();
      const fakeRule = input => ({passed: false,
                                  reason: 'fake reason'});

      verifier.addRule(fakeRule);
      const errors = verifier.verify('any value');

      expect(errors.length).toBe(1);
    });
  });
});
```

This is starting to look *bad*. Yes, we have solved the assertion roulette issue. Each 'it' can fail separately and not interfere with the results from the other test case. But what did it cost? Everything. Look at all the duplication we have now. At this point those of you with some unit testing background will start shouting at the book: "use a `setup` / `beforeEach` method!".

Fine!

2.6 Trying the beforeEach() route

I haven't introduced `beforeEach` yet. This function and its brother, `afterEach()` are used to setup and teardown specific state required by test cases. There's also `beforeAll`*()* and `afterAll`*()* – I try to avoid using them at all costs for unit testing scenarios. We'll talk more about the siblings later in the book.

`beforeEach`() can technically help us remove duplication in our tests because it runs one before each test in the `describe` block in which we nest it under. We can also nest it multiple times:

Listing 2.13 – using beforeEach() on two levels

```
//ch2/__tests__/password.verifier1.spec.js
...
describe('v4 PasswordVerifier', () => {
  let verifier;
  beforeEach(() => verifier = new PasswordVerifier1());
  describe('with a failing rule', () => {
    let fakeRule, errors;
    beforeEach(() => {
      fakeRule = input => ({passed: false, reason: 'fake reason'});
      verifier.addRule(fakeRule);
    });
    it('has an error message based on the rule.reason', () => {
      errors = verifier.verify('any value');

      expect(errors[0]).toContain('fake reason');
    });
    it('has exactly one error', () => {
      const errors = verifier.verify('any value');
17:
18:       expect(errors.length).toBe(1);
19:     });
20:   });
21: });
22:
```

Look at all that deleted code.

In line 3 we're setting up a new `PasswordVerifier1` that will be created for each test case.
In line 6 we're setting up a fake `rule` and adding it to the new verifier, for every test case under that specific scenario. If we had other scenarios, the `beforeEach`() in line 6 wouldn't run for them. But the one in line 3 would.

The tests seem shorter now, which ideally is what you'd want in a test to help make it more readable and maintainable. We removed the creation line from each test and re-using the same higher level variable *verifier.*

A couple of caveats:

1. We forgot to reset the `errors` array in `beforeEach on line 6`. That could bite us later on.
2. Jest by default runs our unit tests in parallel. This means that moving the verifier to line 2 can cause an issue with parallel tests where the verifier could be overwritten by a different tests on a parallel run, and screwing up the state of our running test. Jest is quite unique from other unit test frameworks in most other languages I know, which make a point of running tests in a single thread, not in parallel (at least by default), to avoid such issues. We jest we have to remember parallel tests are a reality, so stateful tests with shared upper state like we have here at line 2 can potentially be problematic in the long run and cause flaky tests that fail for unknown reasons.

We'll both of these out soon. But first…

2.6.1 beforeEach() and scroll fatigue

We did lose a couple of things in the process of refactoring to `beforeEach`():

If I'm trying to read only the '`it`' parts, I can't tell where `verifier` is created and declared. I'll have to scroll up to understand.

The same goes for understanding what rule was added. I'd have to look one level above the '`it`' to see what rule was added, or to look up the '`describe`' block description.

Right now this doesn't seem so bad. But we'll see later that this structure starts to get a bit hairy as the scenario list increases in size. Larger files can bring about what I like to call 'scroll-fatigue' – requiring the test reader to scroll up and down the test file to continuously understand the context and state of the tests. This makes maintaining and reading the tests a chore instead of a simple act of reading.

This nesting is great for reporting, but it sucks for humans who have to keep looking up where something came from.

If you’ve ever tried to debug `CSS` styles in the browser’s inspector window, you’ll know the feeling. You see that a specific cell is **bold** for some reason. Then you scroll up to see which style initially decided that nested cells inside a special `table` under the third node needed to be bold <`div`>

Let’s see what happens when we take it one step further. Since we’re in the process of removing duplication, We can also call *verify* in `beforeEach` and remove an extra line from each ‘`it`’. This is basically putting ‘arrange’ and ‘act’ parts form the ‘arrange, act, assert’ pattern into the `beforeEach` function:

Listing 2.14 – pushing the ‘act’ part into beforeEach()

```
//ch2/__tests__/password.verifier1.spec.js
...
describe('v5 PasswordVerifier', () => {
  let verifier;
  beforeEach(() => verifier = new PasswordVerifier1());
  describe('with a failing rule', () => {
    let fakeRule, errors;
    beforeEach(() => {
      fakeRule = input => ({passed: false, reason: 'fake reason'});
      verifier.addRule(fakeRule);
      errors = verifier.verify('any value');
    });
    it('has an error message based on the rule.reason', () => {
      expect(errors[0]).toContain('fake reason');
    });
    it('has exactly one error', () => {
      expect(errors.length).toBe(1);
    });
  });
});
```

The code duplication has been reduced to a minimum. But now we also need to look up *where and how* we got the *errors* array if we want to understand each ‘`it`’.

Let’s double down and add a few more basic scenarios and see if this approach is scalable as the problem space increases:

Listing 2.15 – Adding extra scenarios

```
//ch2/__tests__/password.verifier1.spec.js
...
describe('v6 PasswordVerifier', () => {
  let verifier;
  beforeEach(() => verifier = new PasswordVerifier1());
  describe('with a failing rule', () => {
    let fakeRule, errors;
    beforeEach(() => {
      fakeRule = input => ({passed: false, reason: 'fake reason'});
      verifier.addRule(fakeRule);
      errors = verifier.verify('any value');
    });
    it('has an error message based on the rule.reason', () => {
      expect(errors[0]).toContain('fake reason');
    });
    it('has exactly one error', () => {
      expect(errors.length).toBe(1);
    });
  });
  describe('with a passing rule', () => {
    let fakeRule, errors;
    beforeEach(() => {
      fakeRule = input => ({passed: true, reason: ''});
      verifier.addRule(fakeRule);
      errors = verifier.verify('any value');
    });
    it('has no errors', () => {
      expect(errors.length).toBe(0);
    });
  });
  describe('with a failing and a passing rule', () => {
    let fakeRulePass,fakeRuleFail, errors;
    beforeEach(() => {
      fakeRulePass = input => ({passed: true, reason: 'fake success'});
      fakeRuleFail = input => ({passed: false, reason: 'fake reason'});
      verifier.addRule(fakeRulePass);
      verifier.addRule(fakeRuleFail);
      errors = verifier.verify('any value');
    });
    it('has one error', () => {
      expect(errors.length).toBe(1);
    });
    it('error text belongs to failed rule', () => {
      expect(errors[0]).toContain('fake reason');
    });
  });
});
```

Do we like this? I don't. Now we're seeing a couple of extra problems:

I can already start to see lots of repetition in the `beforeEach` parts.

Scroll-fatigue potential has increased dramatically, with more options of which `beforeEach` affects which '`it`' state.

As an extra anti-pattern, in real projects, `beforeEach`() functions tend to be the 'garbage-bin' of the test file. People throw all kinds of test initialized stuff in there. Things that only some tests need, things that affect all the other tests, and things that nobody every uses anymore. It's human nature to put things in the easiest place possible, especially if everyone else before you have done so as well.

I'm not crazy about `beforeEach`. Let's see if we can mitigate some of these issues, while still keeping duplication to a minimum.

2.7 Trying the factory method route

Factory methods are simple helper functions that help us build objects or special state and reusing the same logic in multiple places. Perhaps we can reduce some of this duplication and clunky feeling by using a couple of factory methods for the failing and passing rules:

Listing 2.16 – adding a couple of factory methods ot the mix

```
//ch2/__tests__/password.verifier1.spec.js
...
File: somefile.js
describe('v7 PasswordVerifier', () => {
  let verifier;
  beforeEach(() => verifier = new PasswordVerifier1());
  describe('with a failing rule', () => {
    let errors;
    beforeEach(() => {
      verifier.addRule(makeFailingRule('fake reason'));
      errors = verifier.verify('any value');
    });
    it('has an error message based on the rule.reason', () => {
      expect(errors[0]).toContain('fake reason');
    });
    it('has exactly one error', () => {
      expect(errors.length).toBe(1);
    });
  });
  describe('with a passing rule', () => {
    let errors;
    beforeEach(() => {
      verifier.addRule(makePassingRule());
      errors = verifier.verify('any value');
    });
    it('has no errors', () => {
      expect(errors.length).toBe(0);
    });
  });
  describe('with a failing and a passing rule', () => {
    let errors;
    beforeEach(() => {
      verifier.addRule(makePassingRule());
      verifier.addRule(makeFailingRule('fake reason'));
      errors = verifier.verify('any value');
    });
    it('has one error', () => {
      expect(errors.length).toBe(1);
    });
    it('error text belongs to failed rule', () => {
      expect(errors[0]).toContain('fake reason');
    });
  });

…
  const makeFailingRule = (reason) => {
    return (input) => {
      return { passed: false, reason: reason };
    };
  };
  const makePassingRule = () => (input) => {
    return { passed: true, reason: '' };
  };
})
```

2.7.1 Replacing beforeEach() completely with factory methods.

What if we don’t use `beforeEach` to initialize various things? What if we switched to using small factory methods instead? Let’s see what that looks like:

Listing 2.17 – replacing beforeEach with factory methods

```
//ch2/__tests__/password.verifier1.spec.js
...
const makeVerifier = () => new PasswordVerifier1();
const passingRule = (input) => ({passed: true, reason: ''});

const makeVerifierWithPassingRule = () => {
  const verifier = makeVerifier();
  verifier.addRule(passingRule);
  return verifier;
};

const makeVerifierWithFailedRule = (reason) => {
  const verifier = makeVerifier();
  const fakeRule = input => ({passed: false, reason: reason});
  verifier.addRule(fakeRule);
  return verifier;
};

describe('v8 PasswordVerifier', () => {
  describe('with a failing rule', () => {
    it('has an error message based on the rule.reason', () => {
      const verifier = makeVerifierWithFailedRule('fake reason');
      const errors = verifier.verify('any input');
      expect(errors[0]).toContain('fake reason');
    });
    it('has exactly one error', () => {
      const verifier = makeVerifierWithFailedRule('fake reason');
      const errors = verifier.verify('any input');
      expect(errors.length).toBe(1);
    });
  });
  describe('with a passing rule', () => {
    it('has no errors', () => {
      const verifier = makeVerifierWithPassingRule();
      const errors = verifier.verify('any input');
      expect(errors.length).toBe(0);
    });
  });
  describe('with a failing and a passing rule', () => {
    it('has one error', () => {
      const verifier = makeVerifierWithFailedRule('fake reason');
      verifier.addRule(passingRule);
      const errors = verifier.verify('any input');
      expect(errors.length).toBe(1);
    });
    it('error text belongs to failed rule', () => {
      const verifier = makeVerifierWithFailedRule('fake reason');
      verifier.addRule(passingRule);
      const errors = verifier.verify('any input');
      expect(errors[0]).toContain('fake reason');
    });
  });
});
```

The length is about the same as the previous example, but I find the code to be more readable, and thus, more easily maintained. We've eliminated the `beforeEach` functions, but we did not lose maintainability. The amount of repetition is negligible, but the readability has improved greatly, due to the removal of nested `beforeEach` blocks.

Furthermore, we've reduced scroll-fatigue risk. As a reader of the test, I don't have to scroll up and down the file to find out when an object is created or declared. I can glean all the information from the 'it'. We don't need to know 'how' something is created, but we know *when* it is created sand with what important parameters it is initialized. Everything is *explicitly* explained.

If the need arises, I can drill into specific factory methods, and I like that each '`it`' is self-encapsulating its own state. The nested `describe` structure acts as a good way to know where we are, but the state is all triggered from inside the 'it' blocks. Not outside of them.

2.8 Going Full Circle to test()

In fact, the tests in listing 2.16 are self-encapsulated enough, that the `describes` act only as an added sugar for understanding. They are no longer needed if we don't want them. We could write the tests like this if we wanted to:

Listing 2.18 – removing nested describes

```
//ch2/__tests__/password.verifier1.spec.js
...
// v9 tests
test('pass verifier, with failed rule, ' +
          'has an error message based on the rule.reason', () => {
  const verifier = makeVerifierWithFailedRule('fake reason');
  const errors = verifier.verify('any input');
  expect(errors[0]).toContain('fake reason');
});
test('pass verifier, with failed rule, has exactly one error', () => {
  const verifier = makeVerifierWithFailedRule('fake reason');
  const errors = verifier.verify('any input');
  expect(errors.length).toBe(1);
});
test('pass verifier, with passing rule, has no errors', () => {
  const verifier = makeVerifierWithPassingRule();
  const errors = verifier.verify('any input');
  expect(errors.length).toBe(0);
});
```

```
test('pass verifier, with passing  and failing rule,' +
     ' has one error', () => {
 const verifier = makeVerifierWithFailedRule('fake reason');
 verifier.addRule(passingRule);
 const errors = verifier.verify('any input');
 expect(errors.length).toBe(1);
});
test('pass verifier, with passing  and failing rule,' +
     ' error text belongs to failed rule', () => {
 const verifier = makeVerifierWithFailedRule('fake reason');
 verifier.addRule(passingRule);
 const errors = verifier.verify('any input');
 expect(errors[0]).toContain('fake reason');
});
```

The factory methods provide us all the functionality we need, without losing clarity for each specific test. I kind of like the terseness of listing 2.17. It's easy to understand. We might lose a bit of structure clarity here. So there are instances where I go with the `describe`-less approach, and places where nested describes make things more readable. The sweet spot of maintainability/readability for your project is probably somewhere between these two points.

2.9 Refactoring to parameterized tests

Now, let's move away from the verifier class work on creating and testing a new custom rule for the verifier. Here's a simple one for an uppercase letter (I realize passwords with these requirements are no longer considered a great idea, but for demonstration purposes I'm OK with it):

Listing 2.19 – Password Rules

```
//password-rules.js
const oneUpperCaseRule = (input) => {
  return {
   passed: (input.toLowerCase() !== input),
   reason: 'at least one upper case needed'
  };
 };

 module.exports = {
  oneUpperCaseRule
 };
```

We could write a couple of tests that look like this:

Listing 2.20 – testing a rule with variations

```
// password-rules.spec.js
const { oneUpperCaseRule } = require('../password-rules');
describe('one uppercase rule', function () {
  test('given no uppercase, it fails', () => {
   const result = oneUpperCaseRule('abc');
   expect(result.passed).toEqual(false);
```

```
    });
    test('given one uppercase, it passes', () => {
      const result = oneUpperCaseRule('Abc');
      expect(result.passed).toEqual(true);
    });
    test('given a different uppercase, it passes', () => {
      const result = oneUpperCaseRule('aBc');
      expect(result.passed).toEqual(true);
    });
  });
```

In the listing above, I've highlighted some duplication we might have if we're trying out the same scenario with small variations in the inputs to the unit of work. In this case, we want to test that it should not matter where the uppercase is, as long as it's there. But the duplication will hurt us down the road if we ever want to change the uppercase logic, or need to correct the assertions in some way for that use case.

There are a few ways to create parametrized tests in JavaScript.

, and jest already includes one that's built in with `test.each` (also aliased to `it.each`). Here's how we could use this feature to remove duplication in our tests:

Listing 2.21 – using test.each

```
// password-rules.spec.js
...
01: describe('v2 one uppercase rule', () => {
02:   test('given no uppercase, it fails', () => {
03:     const result = oneUpperCaseRule('abc');
04:     expect(result.passed).toEqual(false);
05:   });
06:
07:   test.each(['Abc',
08:               'aBc'])
09:     ('given one uppercase, it passes', (input) => {
10:       const result = oneUpperCaseRule(input);
11:       expect(result.passed).toEqual(true);
12:     });
13: });
```

- In line 7 we're passing in an array of values. These will be mapped to the first parameter 'input' in our test function.
- In line 9 we are using each input parameter passed in the array
- The test will repeat once per value in the array.

It's a bit of a mouthful at first, but once you've done this once it becomes easy to repeat.

It's also pretty readable.

If we want to pass in multiple parameters, we can enclosed them in an array, as we'll see in the next example.

Listing 2.22 – refactoring test.each

```
// password-rules.spec.js
```

```
...
01: describe('v3 one uppercase rule', () => {
02:   test.each([ ['Abc', true],
03:               ['aBc', true],
04:               ['abc', false]])
05:     ('given %s, %s ', (input, expected) => {
06:       const result = oneUpperCaseRule(input);
07:       expect(result.passed).toEqual(expected);
08:     });
09: });
```

In this example we've added a new parameter to our test, with the expected value.

- In lines 2-4 we are providing three arrays, each with two parameters.
- Line 4 adds a 'false' expectation for a missing upper case
- In line 5 we are putting the value of each parameters in the string using %s formatting functions inside the test name. Jest maps the values automatically for us.
- Line 5 also see jest mapping the two parameters for us into the test callback.

We don't have to use Jest, though. Javascript is versitile enough ot allow to roll out our own parameterized test quite easily if we wanted to:

Listing 2.23 – using test.each

```
describe('v5 one uppercase rule, with vanila JS test.each', () => {
  const tests = {
    'Abc': true,
    'aBc': true,
    'abc': false,
  };

  for (const [input, expected] of Object.entries(tests)) {
    test(`given ${input}, ${expected}`, () => {
      const result = oneUpperCaseRule(input);
      expect(result.passed).toEqual(expected);
    });
  }
});
```

It's up to you which one you'd want to use (I like to keep it simple and use `test.each`). The point is, Jest is just a tool. The pattern of parameterized tests can be implemented in multiple ways. This pattern gives us a lot of power, but also a lot of responsibility. It is really easy to abuse this technique and create tests that are harder to understand.

I usually try to make sure that the same scenario (type of input) holds for the entire table.

If I were reviewing this test in a code review, I would have told the person who wrote it that this test is actually testing two different scenarios: One with *no uppercase*, and a couple with *one uppercase*. I would split those out into two different tests.

In this example I wanted to show that it's very easy to get rid of many tests and put them all in a big `test.each` – even when it hurts readability – so be careful when running with these specific scissors.

2.10 Checking for expected thrown errors

Sometimes we need to design a piece of code that throws an error at the right time with the right data. if we added code to the verify function that throws an error if there are no rules configured, like so?

Listing 2.24 – throwing an error

```
// password-verifier1.js

verify (input) {
    if (this.rules.length === 0) {
      throw new Error('There are no rules configured');
    }
    …
```

We could test it the old fashioned way by using `try-catch`, and failing the test if we *didn't get an error*, like so:

Listing 2.25 – testing exceptions with try-catch

```
ch2/__tests__/password.verifier1.spec.js

test('verify, with no rules, throws exception', () => {
    const verifier = makeVerifier();
    try {
        verifier.verify('any input');
        fail('error was expected but not thrown');
    } catch (e) {
        expect(e.message).toContain('no rules configured');
    }
});
```

Using `fail()`

Technically, `fail()` is a leftover api from the original fork of `Jasmine` which Jest is based on. It's a way to trigger a test failure. It's not in the official jest API docs, and they would recommend that you use `expect.assertions(1)` instead. -this would fail the test if you never reached the catch() expectation. I find that as long as `fail()` still works, it does the job quite nicely for my purposes, which are to demonstrate why you shouldn't use the try-catch construct anyway in a unit test if you can help it.

This try-catch pattern is an effective method but very verbose and annoying to type. Jest, like most other frameworks, contains a shortcut to accomplish exactly this type of scenario, using `expect().toThrowError()`:

Listing 2.26 – using expect.toThrowError

```
ch2/__tests__/password.verifier1.spec.js

test('verify, with no rules, throws exception', () => {
    const verifier = makeVerifier();
3:     expect(() => verifier.verify('any input'))
4:         .toThrowError(/no rules configured/);
5: });
```

Notice that in line 4 I'm using a regular expression match to check that the error string *contains* a specific string ,and is not equal to it, so as to make the test a bit more future proof if the string changes on its sides. `toThrowError` has a few variations, and you should go to `https://jestjs.io/` find out all about them.

Jest Snapshots

Jest has a unique feature called snapshots. It allows one to render a component (when working in a framework like React) and then match the current rendering to a saved snapshot of that component, including all of its properties and html.

I won't be touching on this too much, but from what I've seen (as well as some of the reviewers of this book have noted) this feature tends to be abused quite heavily. You can use it to create hard to read tests that look somewhat like this:

```
it('renders',()=>{
    expect(<MyComponent/>).toMatchSnapshot();
});
```

This is both obtuse (hard to reason about what is being tested) and it's testing many things that might not be related to one another. It will also break for many reasons that you might no care about, so the maintainability cost of that test will be higher over time. It's also a great excuse not to write readable maintainable tests, because you're on a deadline but still have to show you write tests. Yes, it does serve a purpose, but it's easy to use in places where other types of tests are more relevant.

If you need a variation of this, try to use the `toMatchInlineSnapshot()` instead.

You can find more info at https://jestjs.io/docs/en/snapshot-testing

2.11 Setting Test Categories

If you'd like to run only a specific 'category' of tests, say, only unit tests, or only integration tests, or only tests that touch a specific part of the application, jest currently doesn't seem to have the ability to define testcase categories.

All is not lost, though. Jest has a special command line flag called `--testPathPattern` which allows us to define how jest will find our tests. We can trigger this command with a different path for a specific type of test we'd like to run (i.e "all tests under 'integration' folder").

You can get the full details on it over at https://jestjs.io/docs/en/cli

Another alternative is to create separate `jest.config.js` file per test category, each with its own `testRegex` configuration and other properties:

Listing 2.26 – removing nested describes

```
// jest.config.integration.js
var config = require('./jest.config')
config.testRegex = "integration\\.js$"
module.exports = config

// jest.config.unit.js
var config = require('./jest.config')
config.testRegex = "unit\\.js$"
module.exports = config
```

Then, for each category you can create a separate npm script that invokes the jest command line with a custom config file: `jest -c my.custom.jest.config.js` :

Listing 2.27 – removing nested describes

```
//Package.json
...
"scripts": {
      “unit”: “jest -c jest.config.unit.js”,
      “integ”: “jest -c jest.config.integration.js”
...
```

2.12 Summary

We’ve covered quite a bit here:

- Jest basics (`Describes`, `its` and `tests`)
- Refactoring and `beforeEach` – when it makes sense, and how it can hurt readability and scroll fatigue
- Factory methods & Parameterized tests can help maintainability even more
- Testing error throwing functionality

Jest has plenty of syntax sugar. You can read about all the assertions I don’t plan to cover as they are not material to this book, over here: https://jestjs.io/docs/en/expect

In the next chapter, we’ll look at code that has dependencies and testability problems, and we’ll start discussing the idea of fakes, spies, mocks, and stubs, and how you can use them to write tests against such code.

3 Breaking dependencies with stubs

This chapter covers

- **Types of Dependencies – Mocks, Stubs & more**
- **Reasons to use stubs**
- **Functional Injection Techniques**
- **Object Oriented Injection Techniques**

In the previous chapter, you wrote your first unit test using jest, and we looked more at the possible structure at maintainability of the test itself. The scenario was pretty simple, and more importantly, it was completely self-contained. The password verifier had no reliance on outside modules, and we could focus only on its functionality without worrying about other things that might interfere with it.

We used the first two types of *exit points* for our examples: **Return value** exit points, and **state-based** exit points. In this chapter we'll talk about the final one – **calling a 3rd party**.

This chapter will present a new requirement: – having your code rely on time. We'll look at two different approaches to handling it – refactoring our code, and monkey-patching it without refactoring.

The reliance on outside modules or functions can and will make it harder to write the test, make the test repeatable, and can cause tests to be flaky.

We call those external *things* that we rely on in our code ***dependencies***. I'll define them more thoroughly later in the chapter. These could include things like time, async execution, using the file system, using the network, or it could simply be using something that is very difficult to configure or may be time consuming to execute.

3.1 Types of Dependencies

In my experience there are two main types of dependencies that our unit of can use.

***Outgoing Dependencies*:** dependencies that represent an *exit point* of our unit of work. For example: calling a logger, saving something to a database, sending an email, notifying an API a webhook that something has happened etc. Notice these are all *verbs*: "calling", "sending" and "notifying". They are flowing *outward* from the unit of work in a sort of "fire and forget" scenario. They represent an exit point, or the end of a specific logical flow in a unit of work.

Incoming Dependencies: dependencies that are not exit points. They do not represent a requirement on the eventual behavior of the unit of work. They are merely there to provide test-specific specialized data or behavior into the unit of work. For example: a database query's result, the contents of a file on the filesystem, a network response, etc. Notice these are all passive pieces of data that flow *inward* into the unit of work as a result of a previous operation.

Figure 3.1 shows these two side by side:

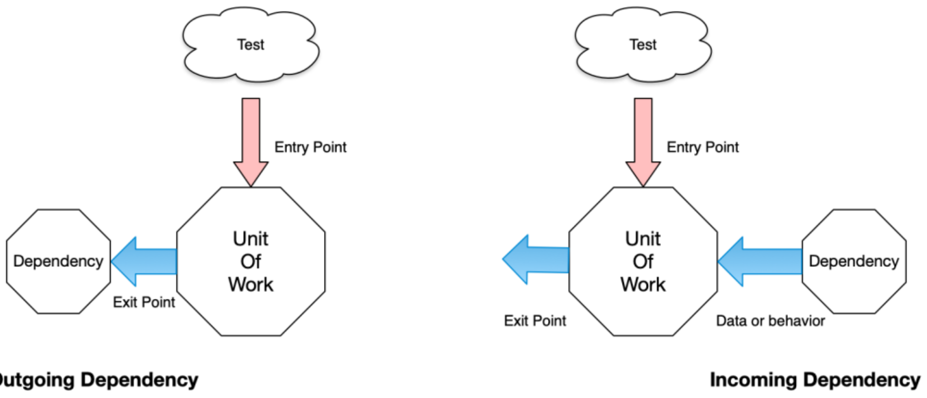


Figure 3.1: On the left – an exit point that is implemented as invoking a dependency. On the right- Dependency provides indirect input or behavior and is not an exit point.

Some dependencies can be both incoming and outgoing. Which means in some tests they will represent exit points, and in other tests they will be used to simulate data coming into the application. These shouldn't be very common, but they do exist. For example, an external API that returns a success/fail response for an outgoing message.

With these types of dependencies in mind, let's look at how the book "XUnit Test Patterns" defines the various patterns for things that look like other things in tests.

The following table combines my thoughts and some of the content from the book's website at http://xunitpatterns.com/Mocks,%20Fakes,%20Stubs%20and%20Dummies.html

Category	Pattern	Purpose	Usages
	Test Double	Generic name for family	(I also use the term

			‘fake’)
Stub	**Dummy Object**	*How do we specify the values to be used in tests when the only usage is as irrelevant arguments of SUT method calls?*	Send as a parameter to the entry point or as part of the ‘arrange’.
	Test Stub	*How can we verify logic independently when it depends on indirect inputs from other software components?”*	Inject as a dependency, and configure to return specific values or behavior into the SUT
Mock	**Test Spy**	*How can we verify logic independently when it has indirect outputs to other software components?*	Override a single function on a real object, and verify the fake function was called as expected.
	Mock Object	*How can we verify logic independently when it depends on indirect inputs from other software components?*	Inject fake as a dependency into the SUT, and verify that fake was called as expecrted

Here’s another way to think about this for the rest of this book:

STUB: Break incoming dependencies (indirect inputs). Stubs are fake modules, objects or functions that provide fake behavior or data *into* the code under test. We do not assert against them. We can have many stubs in a single test.

MOCK: Break outgoing dependencies (outputs, exit points). Mocks are fake modules, objects or functions that we assert were called in our tests. A Mock represents an *exit point* in a unit test. Because of this, it is recommended to have no more than a single mock per test.

Unfortunately, today in many shops you’ll hear the word “mock” thrown around as a catch-all term for both stubs and mocks. “We’ll mock this out”, “we have a mock database” and more are phrases that really can create confusion. There is a huge difference between stub and mock (one should really only be used once in a test). We should use the right term to remove implicit assumptions about what the other person is referring to.

When in doubt, use “test double” or “fake”. Many times, the same fake dependency in one test is used as a stub, and can be used as a mock in another test. We’ll see an example of that later on.

XUnit Test Patterns and Naming Things

xUnit Test Patterns: Refactoring Test Code by Gerard Meszaros (Addison-Wesley, 2007) is a classic pattern reference book for unit testing. It defines patterns for things you fake in your tests in at least five ways. Once you've gotten a feel for the three types I mention here, I encourage you to take a look at the extra details that book provides.

Do note that that book has a definition for the word "Fake" :

"Replace a component that the system under test (SUT) depends on with a much lighter-weight implementation."

For example , an in memory database instead of a full fledged production instance.

I still consider this type of test double a "stub" , and I use the word "fake" to just call out anything that isn't real, much like "test double" - but it's shorter and easier on the toungue.

This all might seem like a whole lot of information at once. I'll deep dive into these definitions throughout this chapter. Let's take a small bite and start with *stubs*.

3.2 Reasons to use stubs

What if we're faced with the task of testing the piece of code like the one in listing 3.1?

Listing 3.1 verifyPassword using time

```
File: ch3/stub-time/00-parameters/password-verifier-time00.js

const moment = require('moment');
const SUNDAY = 0, SATURDAY = 6;

const verifyPassword = (input, rules) => {
    const dayOfWeek = moment().day();
    if ([SATURDAY, SUNDAY].includes(dayOfWeek)) {
        throw Error("It's the weekend!");
    }
    //more code goes here...
    //return list of errors found..
    return [];
};
```

out password verifier has a new dependency: it can't work on weekends. Go figure. :

```
const dayOfWeek = moment().day();
```

Specifically, the module has a direct dependency on moment.js, which is a very common date/time wrapper for JavaScript. Working with dates directly in JavaScript is not a pleasant experience, so we can assume many shops out there have something like this.

How does this direct usage of a time related library affect our unit tests?

The unfortunate issue here is that this direct dependency forces our tests, given no direct way to affect date and time inside our application under test, to also take into account the correct date and time.

Listing 3.2 shows an unfortunate test that only runs on weekends:

Listing 3.2 verifyPassword – initial unit tests

```
File: ch3/stub-time/00-parameters/password-verifier-time00.spec.js

01: const moment = require('moment');
02: const {verifyPassword} = require("./password-verifier-time00");
03: const SUNDAY = 0, SATURDAY = 6, MONDAY = 2;
04:
05: describe('verifier', () => {
06:     const TODAY = moment().day();
07:
08:     //test is always executed, but might not do anything
09:     test('on weekends, throws exceptions', () => {
10:         if ([SATURDAY, SUNDAY].includes(TODAY)) {
11:             expect(()=> verifyPassword('anything',[]))
12:                 .toThrowError("It's the weekend!");
13:         }
14:     });
15:
16:     //test is not even executed on week days
17:     if ([SATURDAY, SUNDAY].includes(TODAY)) {
18:         test('on a weekend, throws an error', () => {
19:             expect(()=> verifyPassword('anything', []))
20:                 .toThrow("It's the weekend!");
21:         });
22:     }
23: });
24:
25:
```

I’ve put in two variations on the same test. One that checks for the current date inside the test (line 10), and the other has the check outside the test (line 17), which means the tests never even executes unless it’s the weekend. This is bad, and you should feel bad.

Let’s revisit one of the good test qualities mentioned in chapter 1:

CONSISTENT: **Every time I run the test it is the *same exact test* that had run before. The values being used do not change. The asserts do not change. If no code has changed (in tests or production code) then the test should provide the same exact result as previous runs.**

This test sometimes doesn’t even run. That’s a good enough reason for use to break the dependency right there. Let’s add to that the fact that we cannot simulate a weekend or a weekday and we have more than enough incentive to go ahead and redesign the code under test to be a bit more “injectable” for dependencies.

But wait, there’s more. Tests that use time, can often be “flaky”. They only fail sometimes, without anything else but the time changing. This test is a prime candidate for this behavior because we’ll only get feedback on *one* of its two states when we run it locally. Want ot know how it behaves on a weekend, just wait a couple of days. Ugh.

Tests might become “flaky” sometimes due to edge cases that occur which affect variables that are not under our control in the test. Common examples of those are network issues during end-to-end testing, database connectivity issues, servers being down, or various server issues.

When this happens, it’s easy to dismiss the test failure by saying “just run it again” or “it’s OK. It’s just [insert variability issue here]”

3.3 Generally accepted design approaches to stubbing

In the next few sections we’ll discuss several common forms of injecting stubs into our unit of work. First we’ll discuss basic parameterization as a first step, then we’ll jump into:

Functional approaches

- Function as Parameter
- Factory Functions (a.k.a Higher Order Functions)
- Constructor Functions

Object Oriented Approaches

- Class Constructor Injection
- Object as Parameter (a.k.a ‘duck typing’)
- Common Interface as Parameter (for this we’ll use TypeScript)

We will tackle each of these by starting with the simple case of controlling time in our tests.

3.3.1 Stubbing out time with parameter injection

I can think of at least two good reasons to control time based on what we covered so far:

1. To be able to remove the variability from our tests
2. To be able to simulate easily any time related scenario we’d like to test our code with

Here’s the simplest refactoring I can think of to make things a bit more repeatable:

Let’s a add a parameter to our function to take in the date from which we can search. This would remove the need to use the current date and time from our function and put the responsibility on the caller of the function. That way, in our tests we can determine the time in a hardcoded manner and make the test and the function repeatable and consistent.

Listing 3.3 shows an example of such a refactoring.

Listing 3.3 verifyPassword with a parameter currentDay

```
File: ch3/stub-time/00-parameters/password-verifier-time00.js
-----------------------------------------------------------------

const verifyPassword2 = (input, rules, currentDay) => {
    if ([SATURDAY, SUNDAY].includes(currentDay)) {
        throw Error("It's the weekend!");
    }
    //more code goes here...
    //return list of errors found..
    return [];
};

File: ch3/stub-time/00-parameters/password-verifier-time00.spec.js
-----------------------------------------------------------------
```

```
const {verifyPassword2} = require("./password-verifier-time00");
const SUNDAY = 0, SATURDAY = 6, MONDAY = 2;
describe('verifier2 - dummy object', () => {
    test('on weekends, throws exceptions', () => {
        expect(()=> verifyPassword2('anything',[],SUNDAY ))
            .toThrowError("It's the weekend!");
    });
});
```

By adding the `currentDay` parameter, we're essentially giving the control over time to the caller of the function (our test). What we're injecting is formally called a "dummy" – it's just a piece of data with no behavior. But we can just call it a stub from now on.

This is also known as a form of "Dependency Inversion". It's not totally clear, but it seems the term "Inversion of Control" first came to light in Johnson and Foote's paper "Designing Reusable Classes", published by the Journal of Object-Oriented Programming in 1988. The term "Dependency Inversion" is also one of the patterns described in the SOLID design principles paper and ideas from Robert C. Martin (I'll talk more about the higher-level design considerations later in the book, in the "Design and Testability" chapter.

It's a simple refactoring, but quite effective. it provides a couple of nice benefits other than just consistency in the test:

- We can now easily simulate any day we want
- The code under test is not responsible for managing time imports, so it has one less reason to change if we ever use a different time library.

We're doing "dependency injection" of time into our unit of work. This is enabled because we've changed the design of the entry point or its context (in a class or higher order function) to use a day value or a day function as a parameter. The function is more "pure" by functional programming standards, in that it has no side effects. Pure functions have built in injections of all of their dependencies. That's also one of the reason's you'll find functional programming designs are typically much easier to test.

It might feel weird to call it a stub if it's just a day integer value, but based on the defintions from xUnit Test Patterns, we can say this is a 'dummy' value, and that, as far as I'm concerned, falls into the 'stub' category, but it provides a *direct input into the entry point*. It does not have to be "fake" in order to be a stub. It just has to be under our control. It is a stub because we are using it to simulate some input or behavior *into* the unit under test.

Figure 3.2 shows this a bit more visually:

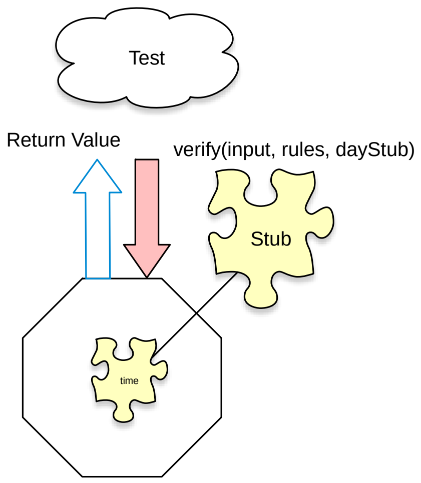


Figure 3.2: Injecting a stub for a time dependency

3.3.2 Dependencies, Injections and Control

Here's a little checkpoint to recap some important terms we've discussed and are about to use throughout the rest of the chapter:

Dependencies	The things that make our testing lives & code maintainability difficult, since we cannot **control** them from our tests. Examples include Time, file system, network, random values and more.
Control	The ability to instruct a dependency how to behave. Whoever is *creating* the dependencies is said to be in control over them since they have the ability to configure them before they are used in the code under test. In listing 3.1 our test does *not* have control of *time* because the module under test has control over it. The module has chosen to always use the current date and time. This forces the test to do the exact same thing and thus we lose consistency in our tests. In listing 3.3 we are gaining access to the dependency by **inverting the control** over it (date parameter). Now the test has control of the time and can decide to use a hardcoded time. The module under test has to use the time provided, which makes things much easier for our test.
Inversion of Control	Designing the code to remove the responsibility of creating the dependency internally and externalizing it instead. Listing 3.3 shows one way of doing it with **parameter injection**.
Dependency Injection	The act of sending a dependency through the design interface to be used internally by apiece of code. The place where you inject the dependency is the injection point. In our case we are using a parameter injection point. Another word for this place where we can inject things is **a seam.**.
Seam	Pronounced “s-ee-m”. Coined by Michael Feathers’ book “Working Effectively with Legacy Code”. Seams are where two pieces of software meet and something else can be injected. They are a place where you can alter behavior in your program without editing in that place. Examples include parameters, functions, module loaders, function rewriting, and in the object oriented world - class interfaces, public virtual methods and more.

Seams in production code play an important role in the maintainability and readability of unit tests. The easier it is to change and inject behavior or custom data into the code under test, the easier it will be to write, read and later on maintain the test as the production code changes. I’ll talk more about some patterns and anti patterns relating to designing the code in the chapter on design and testability.

3.4 Functional Injection Techniques

At this point we might not be happy with our design choice. Adding a parameter did solve the dependency issue at the function level, but now every caller ever would need to know about how to handle dates in some way. It feels a bit to "chatty".

Because JavaScript enables two major styles of programming: functional and object oriented, I'll show approaches in both styles when it makes sense, so that you can pick and choose what works best in your situation.

There isn't a single way to design something. Functional Programming proponents will argue on the simplicity and clarity and provability of a functional style, but it does come with a learning curve that, pragmatically, we have to realize will take more than a couple of years to take in the industry.

For that reason alone, it would be wise to learn both approaches so that you can apply whichever works best *for the team you're working with*. Some teams will lean more towards object oriented designs because they'd feel more comfortable with that. Others will lean towards functional. I'd argue that the patterns remain largely the same, we just translate them to different styles.

3.4.1 Injecting a function

Listing 3.4 shows a different refactoring for the same problem: instead of a data object, we're expecting a function as the parameter. That function returns the date object:

Listing 3.4: Dependency Injection with a function

```
File: ch3/stub-time/00-parameters/password-verifier-time00.js

const verifyPassword3 = (input, rules, getDayFn) => {
    const dayOfWeek = getDayFn();
    if ([SATURDAY, SUNDAY].includes(dayOfWeek)) {
        throw Error("It's the weekend!");
    }
    //more code goes here...
    //return list of errors found..
    return [];
};...
```

Now our test can look like listing 3.5:

Listing 3.5: Testing with function injection

```
File: ch3/stub-time/00-parameters/password-verifier-time00.spec.js

describe('verifier3 - dummy function', () => {
    test('on weekends, throws exceptions', () => {
        const alwaysSunday = () => SUNDAY;
        expect(()=> verifyPassword3('anything',[], alwaysSunday))
            .toThrowError("It's the weekend!");
    });
...
```

There’s very little difference from the previous test but using a function as a parameter is a valid way to do injection. In other scenarios it’s also a great way to enable special behavior such as simulating special cases or exceptions into your code under test.

3.4.2 Using Factory Functions

Factory functions or methods (a subcategory of “higher order functions”) are functions that return other functions, pre-configured with some context. In our case the context can be the list of rules and the current day function. We then get back a new function that we can trigger only with a string input, and it would already be using the rules and `getDay()` function configured in its creation.

This essentially turns the factory function into the “arrange” part in the test, and calling the returned function the “act” part of my test. Quite lovely:

Listing 3.6 Using a higher order factory function

```
File: ch3/stub-time/01-higher-order/password-verifier-time01.js
const SUNDAY = 0, … FRIDAY=5, SATURDAY = 6;

const makeVerifier = (rules, dayOfWeekFn) => {
    return function (input) {
        if ([SATURDAY, SUNDAY].includes(dayOfWeekFn())) {
            throw new Error("It's the weekend!");
        }
        //more code goes here..
    };
};

File: ch3/stub-time/01-higher-order/password-verifier-time01.spec.js
const {makeVerifier} = require("./password-verifier-time01");

describe('verifier', () => {
    test('factory method: on weekends, throws exceptions', () => {
        const alwaysSunday = () => SUNDAY;
        const verifyPassword = makeVerifier([], alwaysSunday);

        expect(()=> verifyPassword('anything'))
            .toThrow("It's the weekend!");
    });
```

3.4.3 Moving Towards Objects with Constructor Functions

Constructor functions are a slightly more “object oriented” JavaScript-ish way of achieving the same result of a factory function, but they return something akin to an object with methods we can trigger. We then use the keyword “new” to call this function and get back that special object.

Here’s how the same code and tests look with this design choice:

Listing 3.7 using a higher order function

```
File: ch3/time/02-callback-injection/machine-scanner2.1.js

File: password-spec.ts
```

```
File: password-spec.ts
// constructor function pattern
const Verifier = function(rules, dayOfWeekFn)
{
    this.verify = function (input) {
        if ([SATURDAY, SUNDAY].includes(dayOfWeekFn())) {
            throw new Error("It's the weekend!");
        }
        //more code goes here..
    };
};
File: password-spec.ts
File: password-spec.ts
//constructor function demo
const {Verifier} = require("./password-verifier-time01");

test('constructor function: on weekends, throws exception‘, () => {
    const alwaysSunday = () => SUNDAY;
    const verifier = new Verifier([], alwaysSunday);

    expect(()=> verifier.verify('anything'))
        .toThrow("It's the weekend!");
});
```

Some of you might look at this and ask "why move towards objects?" and the answer really would depend on the context of your current project, stack, your team's knowledge of functional programming and OO background and many other factors that are not technical by nature. Have this tool in your tool bels so you can use it when it makes sense to you. Keep this in the back of your mind over the next few sections.

3.5 Object Oriented Injection Techniques

If a more object-oriented style is what we're leaning towards, or if you're working in a more object-oriented language such as C# or Java, here are a few common patterns that are widely used in the OO world for dependency injection.

3.5.1 Constructor Injection

Constructor injection is how we would describe a design in which we can inject dependencies through the constructor of a class. In the JS world `Angular` is the most well-known web frontend framework that uses this design for injecting "services" which are just codeword for "dependencies" in angular speak, but it's a viable design in many other situations.

Having a stateful class is not without benefits. It can remove repetition from clients who would only need to configure our class once and then reuse the configured class multiple times.

If we had chosen to create a stateful version of `password verifier` and we wanted to inject the date function through constructor injection, it might look close to this design:

Listing 3.8 constructor injection design

```
File: ch3/stub-time/02-inject-object/password-verifier-time02.js

class PasswordVerifier {
```

```
03:     constructor(rules, dayOfWeekFn) {
04:         this.rules = rules;
05:         this.dayOfWeek = dayOfWeekFn;
06:     }
07:
08:     verify(input) {
09:         if ([SATURDAY, SUNDAY].includes(this.dayOfWeek())) {
10:             throw new Error("It's the weekend!");
11:         }
12:         const errors = [];
13:         //more code goes here..
14:         return errors;
15:     };
16: }

File: ch3/stub-time/02-inject-object/password-verifier-time02.spec.js

1: test('class constructor: on weekends, throws exception', () => {
2:     const alwaysSunday = () => SUNDAY;
3:     const verifier = new PasswordVerifier([], alwaysSunday);
4:
5:     expect(()=> verifier.verify('anything'))
6:         .toThrow("It's the weekend!");
7: });
```

It looks and feels a lot like the "constructor function" design we did earlier. This is a more class-oriented design that many people would feel more comfortable with, coming form an object-oriented background. It also is more verbose. You'll see that we get more and more verbose the more "object-oriented" we make things. It's part of the OO game. This is partly why people are choosing functional styles more and more - they are much more concise.

Let's talk a bit about maintainability of the tests. If I wrote a second test with this class, I'd extract the creation of the class in line 3 to a nice little factory function that returns an instance of the class under test, so that if (read: "when") the constructor signature changes and breaks many tests at once, I only have to fix a single place to get all the tests working again:

Listing 3.9 adding a helper factory function to our tests

```
File: ch3/stub-time/02-inject-object/password-verifier-time02.spec.js

01: describe('refactored with constructor', () => {
02:     const makeVerifier = (rules, dayFn) => {
03:         return new PasswordVerifier(rules, dayFn);
04:     };
05:
06:     test('class constructor: on weekends, throws exceptions', () => {
07:         const alwaysSunday = () => SUNDAY;
08:         const verifier = makeVerifier([],alwaysSunday);
09:
10:         expect(()=> verifier.verify('anything'))
11:             .toThrow("It's the weekend!");
12:     });
13:
14:     test('class constructor: on weekdays, with no rules, passes', () => {
15:         const alwaysMonday = () => MONDAY;
```

```
        const verifier = makeVerifier([],alwaysMonday);

        const result = verifier.verify('anything');
        expect(result.length).toBe(0);
    });
});
...
```

Notice that this is not the same as the "factory function" design we did earlier. This factory function resides in our *tests*. The other was in our production code. This one is for test maintainability and can work with both object-oriented or function style designed production code, because it *hides* how the function or object are being created or configured. It is an abstraction layer in our tests so we can push the *dependency* on how a function or object are created/configured into a single place in our tests.

3.5.2 Injecting an object instead of a function

Right now, our class constructor takes in a function as the second parameter:

```
constructor(rules, dayOfWeekFn) {
this.rules = rules;
this.dayOfWeek = dayOfWeekFn;
}
```

Let's go one step up in our object-oriented design and use an object instead of a function as our parameter. This requires us to do a bit of leg work, refactoring the code:

First, we'll create a new file called `time-provider.js` which will contain our "real" object that has a dependency on `moment.js` . The object will be designed to have a single function called `getDay()`.

```
import moment from "moment";

const RealTimeProvider = () =>  {
    this.getDay = () => moment().day()
};

module.exports = {
    RealTimeProvider
};
```

Next, we'll change the parameter usage to that of using an object with a function

```
const SUNDAY = 0, MONDAY=1, SATURDAY = 6;
class PasswordVerifier {
    constructor(rules, timeProvider) {
        this.rules = rules;
        this.timeProvider = timeProvider;
    }

    verify(input) {
        if ([SATURDAY, SUNDAY].includes(this.timeProvider.getDay())) {
            throw new Error("It's the weekend!");
        }
   ...
}
```

```

module.exports = {
    SUNDAY, MONDAY, SATURDAY,
    PasswordVerifier
};
```

Finally, let’s give whoever needs an instance of our PasswordVerifier the ability to get it preconfigured with the real time provider by default. We’ll do this with a new `passwordVerifierFactory` function that any production code that needs a verifier instance will need to use:

```
const {RealTimeProvider} = require("./time-provider");
const {PasswordVerifier} = require("./password-verifier-time02");

const passwordVerifierFactory = (rules) => {
    return new PasswordVerifier(new RealTimeProvider())
};

module.exports = {
    passwordVerifierFactory
};
```

IoC Containers and Dependency Injection

Of course, there are many other ways to “glue” these two together. I’ve just chosen one to keep things simple. Many frameworks today contain the ability to configure the injection of objects like this, so that we can define how an object is to be constructed. Angular is one such framework.

If you’re using stuff like `Spring Boot` in Java, and `AutoFac` or `StructureMap` in C#, you get the ability to easily configure how to construct objects with constructor injection without needing to create specialized functions for them. Commonly these types of functorialities are called “Inversion of Control (IoC) containers” or “Dependency Injection (DI) Containers”. I’m not using them in this book to avoid creating uneeded extra detail. You don’t need them to create great tests.

In fact, I don’t normally use IoC containers in tests, and will almost always use custom factory functions to inject dependencies. I find that makes my tests my easier to read and reason about.

Even for angular types tests we don’t have to go through Angular’s DI framework to inject a dependency into an object in memory, when we can call that object’s constructor directly and send in fake stuff. As long as we do that in a factory function, we’re not sacrificing maintainability, but also not adding extra code to the tests unless it’s essential to the test.

The following listing shows the entire piece of new code together:

Listing 3.10 Injecting an Object

```
//ch3/stub-time/02-inject-object/time-provider.js
import moment from "moment";

const RealTimeProvider = () =>  {
    this.getDay = () => moment().day()
};

```

```
module.exports = {
    RealTimeProvider
};


// ch3/stub-time/02-inject-object/password-verifier-time02.js
const SUNDAY = 0, MONDAY=1, SATURDAY = 6;
class PasswordVerifier {
    constructor(rules, timeProvider) {
        this.rules = rules;
        this.timeProvider = timeProvider;
    }

    verify(input) {
        if ([SATURDAY, SUNDAY].includes(this.timeProvider.getDay())) {
            throw new Error("It's the weekend!");
        }
        const errors = [];
        //more code goes here..
        return errors;
    };
}

module.exports = {
    SUNDAY, MONDAY, SATURDAY,
    PasswordVerifier
};


// ch3/stub-time/02-inject-object/verifier-factory.js
const {RealTimeProvider} = require("./time-provider");
const {PasswordVerifier} = require("./password-verifier-time02");

const passwordVerifierFactory = (rules) => {
    return new PasswordVerifier(new RealTimeProvider())
};

module.exports = {
    passwordVerifierFactory
};

```

How do we handle this type of design in our tests, where we need to inject a fake object, instead of a fake function? We'll do things in a handwritten way first, so you can see it's not a big deal. Later we can let frameworks help us, but you'll see that sometimes hand-coding fake objects can actually make your test more readable than by using a framework such as `jasmine`, `jest` or `sinon` (we'll cover those later in book when we discuss 'dynamic stubs' and 'dynamic mocks')

First, in our test file, we create a new fake object that has the same function signature as our 'real' time provider, but in reality, it will be controllable by our tests. In this case we'll just use a constructor pattern:

```
function FakeTimeProvider(fakeDay) {
    this.getDay = function () {
        return fakeDay;
    }
}
```

(Note: If you are working in a more object-oriented style you might choose to create a simple class that inherits from a common interface. We'll cover that a bit later in the chapter.)

Next, we'll construct the `FakeTimeProvider` in our tests and inject it into the `verifier` under test:

```
describe('verifier', () => {
    test(‘on weekends, throws exception‘, () => {
        const verifier =
             new PasswordVerifier([], new FakeTimeProvider(SUNDAY));

        expect(()=> verifier.verify('anything'))
            .toThrow("It's the weekend!");
    });
```

Here's how the full test file looks:

Listing 3.11 Creating a hand written stub object

```
File: password-spec.ts
const {SUNDAY, MONDAY} = require("./password-verifier-time02");
const {PasswordVerifier} = require("./password-verifier-time02");


function FakeTimeProvider(fakeDay) {
    this.getDay = function () {
        return fakeDay;
    }
}

describe('verifier', () => {
    test('class constructor: on weekends, throws exception‘, () => {
        const verifier =
new PasswordVerifier([], new FakeTimeProvider(SUNDAY));

        expect(()=> verifier.verify('anything'))
            .toThrow("It's the weekend!");
    });
});

```

This code works because JavaScript by default is a very permissive language. Much like ruby or python, you can get away with 'duck typing' thing. 'Duck Typing' refers to the idea that "if it walks like a duck and it talks like a duck, treat it like a duck". In this case the real object and fake object both implement the same function, even though they are essentially completely different objects. We can simple send one *in place* of the other and the production code should be OK with this.

Of course, we'll only know that this is OK and we didn't make any mistakes or miss anything regarding the function signatures at runtime. If we want to get a bit more confidence we can try things in a more type-safe manner.

3.5.3 Extracting a common interface.

We can take things one step further, and, if we're using Typescript or a strongly typed language such as Java or C# start using interfaces to denote the roles that our dependencies play, and create a 'contract' of sorts that both real objects and fake objects will have to abide by at the compiler level.

First, in a new `time-provider-interface.ts` file (notice this is a typescript file) we'll define our new interface:

```
export interface TimeProviderInterface {
    getDay(): number;
}
```

Second, we define a 'real' time provider that implements our interface in our production code like this:

```
import * as moment from "moment";
import {TimeProviderInterface} from "./time-provider-interface";

export class RealTimeProvider implements TimeProviderInterface {
    getDay(): number {
        return moment().day();
    }
}
```

Third, we define the constructor to our `PasswordVerifier` to take a dependency of our new `TimeProviderInterface` type, instead of having a parameter type of `RealTimeProvider`. We're abstracting away the 'role' of a time provider and declaring that we don't care what is the object being passed, as long as it answers to this role's interface.

```
import {TimeProviderInterface} from "./time-provider-interface";
export const SUNDAY = 0, SATURDAY=6;

export class PasswordVerifier {
    private _timeProvider: TimeProviderInterface;

    constructor(rules: any[], timeProvider: TimeProviderInterface) {
        this._timeProvider = timeProvider;
    }

    verify(input: string):string[] {
        const isWeekened = [SUNDAY, SATURDAY]
            .filter(x=>x=== this._timeProvider.getDay())
            .length>0;

        if (isWeekened) {
            throw new Error("It's the weekend!")
        }
          // more logic goes here
        return [];
    }
}
```

Now, by having a interface to define what a ‘duck’ looks like, we can implement a duck of our own in our tests. It’s going to look a lot like the previous test’s code, but it will have one strong difference: it will be compiler-checked for correctness of the method signatures:

Here’s what our fake time provider looks like in our test file:

```
import {TimeProviderInterface} from "./time-provider-interface";

class FakeTimeProvider implements TimeProviderInterface {
    fakeDay: number;
    getDay(): number {
        return this.fakeDay;
    }
}
```

And our test:

```
describe('password verifier with interfaces', () => {
    test('on weekends, throws exceptions', () => {
        const stubTimeProvider = new FakeTimeProvider();
        stubTimeProvider.fakeDay = SUNDAY;
        const verifier = new PasswordVerifier([], stubTimeProvider);

        expect(()=> verifier.verify('anything'))
            .toThrow("It's the weekend!");
    });
});
```

Listing 3.12 shows the entire code together:

Listing 3.12: Extracting a common interface – production code

```
File:/ch3/stub-time/03-ts-inject-interface/time-provider-interface.ts
export interface TimeProviderInterface {  getDay(): number;  }

//ch3/stub-time/03-ts-inject-interface/real-time-provider.ts
import * as moment from "moment";
import {TimeProviderInterface} from "./time-provider-interface";
export class RealTimeProvider implements TimeProviderInterface {
    getDay(): number {
        return moment().day();
    }
}


File:/ch3/stub-time/03-ts-inject-interface/password-verifier-time03.ts
import {TimeProviderInterface} from "./time-provider-interface";
export const SUNDAY = 0, SATURDAY=6;
export class PasswordVerifier {
    private _timeProvider: TimeProviderInterface;

    constructor(rules: any[], timeProvider: TimeProviderInterface) {
        this._timeProvider = timeProvider;
    }

    verify(input: string):string[] {
        const isWeekened = [SUNDAY, SATURDAY]
            .filter(x=>x=== this._timeProvider.getDay())
            .length>0;
        if (isWeekened) {
            throw new Error("It's the weekend!")
        }
        return [];
    }
}


File:/ch3/stub-time/03-ts-inject-interface/password-verifier-time03.spec.ts
import {TimeProviderInterface} from "./time-provider-interface";
import {PasswordVerifier, SUNDAY} from "./password-verifier-time03";

class FakeTimeProvider implements TimeProviderInterface{
    fakeDay: number;
    getDay(): number {
        return this.fakeDay;
    }
}
describe('password verifier with interfaces', () => {
    test('on weekends, throws exceptions', () => {
        const stubTimeProvider = new FakeTimeProvider();
        stubTimeProvider.fakeDay = SUNDAY;
        const verifier = new PasswordVerifier([], stubTimeProvider);

        expect(()=> verifier.verify('anything'))
            .toThrow("It's the weekend!");
    });
});
```

We've now made a full transition from a purely functional design into a strongly typed, object-oriented design. Which one is best for your team and your project – there is no single answer. I'll talk more about design in a later chapter in this book. I wanted to mainly show that whatever is the design you end up choosing, the pattern of injection remains largely the same, it just uses a different vocabulary or language features to be enabled.

Eventually, it is the injection-ability that enables us to simulate things that would be practically impossible to test in real life. And that's where the idea of stubs shines the most.

We can tell our stubs to return fake values or even to simulate exceptions into our code to see how it handles errors happening from dependencies. Injection has made this possible.

Injection has also made our tests more repeatable, consistent and trustworthy - and I'll talk about trustworthiness in the second part of this book.

3.6 Summary

In this chapter we started going over the idea of dependencies and how they can hurt our tests.

- We started using stubs to replace those dependencies, and looked over several different ways of injecting stubs.
- We took a look at both functional and object oriented styles of injection.
- In the next chapter we'll just into mock objects and how they differ from stubs

You can also find more information on stubbing async operations and observables in JavaScript in the appendix of this book "Faking Async in JavaScript".

4 Interaction testing using mock objects

This chapter covers

- **Defining interaction testing**
- **Reasons to use mock objects**
- **Injecting and using mocks**
- **Functional examples**
- **Modular examples**
- **Object-oriented examples**
- **Dealing with Complicated Interfaces**
- **Partial Mocks**

In the previous chapter, you solved the problem of testing code that depends on other objects to run correctly. You used stubs to make sure that the code under test received all the inputs it needed so that you could test the unit of work in isolation.

Also, so far, you've only written tests that work against the first two of the three types of exit points a unit of work can have: **returning a value** and **changing the state** of the system (you can read more about these in chapter 1).

In this chapter, we'll look at how you test the third type of exit point—a call to a third-party function, module or object. This is important because many times we'll have code that depends on things we can't control. Knowing how to check that type of code is an important skill in the world of unit testing. Basically, we'll try to see how many ways we can find to prove that our unit of work ends up calling a function that we don't control, and what values were sent as arguments.

Using the approaches we've learned so far won't do here, because 3rd party functions usually don't have specialized APIs that allow us to check if they were called correctly. Instead, they internalize their operations for clarity and maintainability. So, how do you test that your unit of work interacts with other 3rd parties correctly? You use mocks.

4.1 Interaction Testing, Mocks and Stubs

Let's define interaction testing:

DEFINITION ***Interaction testing*** **is checking how a unit of work interacts and sends messages (i.e calls functions) to a dependency beyond its control. Mock functions or objects are used to assert that a call was made correctly to an external dependency.**

Let's recall the differences between mocks and stubs as we covered them in chapter 3. The main difference is in the flow of information.

Figure 4.1 shows these two side by side:

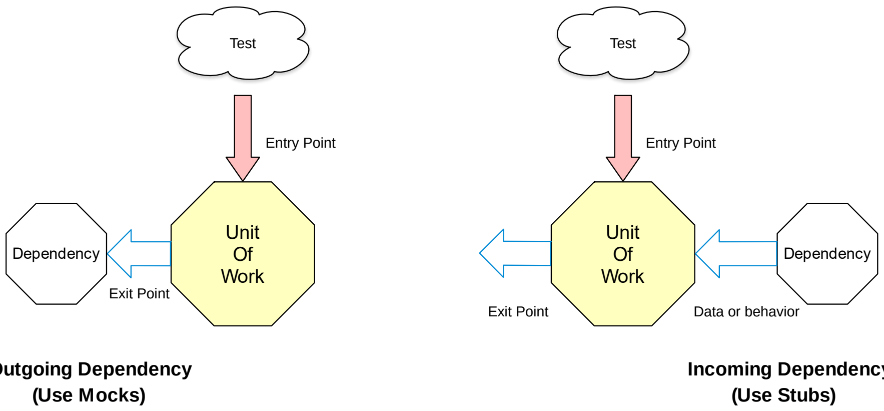


Figure 4.1: On the left – an exit point that is implemented as invoking a dependency. On the right- Dependency provides indirect input or behavior and is not an exit point.

STUB: Used to break incoming dependencies. Stubs are fake modules, objects or functions that provide fake behavior or data *into* the code under test. We do not assert against them. We can have many stubs in a single test.

Stubs represent waypoints, not exit points. They do not represent exit points because the data or behavior flow *into* the unit of work. They are points of interaction, but they do not represent an ultimate outcome the the unit of work ends up doing. Instead they are an interaction *on the way* to achieve the end result we care about, so we don't treat them as exit.

MOCK: Used to break outgoing dependencies. Mocks are fake modules, objects or functions that we assert were called in our tests. A Mock represents an *exit point* in a unit test. If we don't assert on it, it's not used as a mock.

It is normal to have no more than a single mock per test for maintainability and readability reasons (we'll discuss this more in part 3 of this book about writing maintainable tests)

Let's look at a simple example of an exit point to a dependency that we do not control: calling a logger.

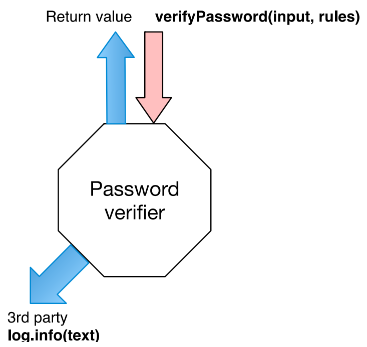
4.2 Depending on a logger

Let's take this **password verifier** function as our starting example. Assuming we have a complicated-logger (A "complicated" logger just has more functions and parameters, so the interface may present more of a challenge for us) part of the requirements for our function, is to call the logger when verification has passed or failed. This is what the code currently looks like:

Listing 4.1: Depending complicated-logger directly

```
File: ch4/00-function-param-injection/00-password-verifier00.js

// impossible to fake with traditional injection techniques
const log = require('./complicated-logger');

const verifyPassword = (input, rules) => {
  const failed = rules
    .map(rule => rule(input))
    .filter(result => result === false);
  if (failed.count === 0) {
    // to test with traditional injection techniques
    log.info('PASSED'); // ← exit point
    return true; // ← exit poit
  }
  //impossible to test with traditional injection techniques
  log.info('FAIL'); // ← exit poinnt
  return false; // ← exit point
};

//////////////////////////////////
File: complicated-logger.js
const info = (text) => {
    console.log(`INFO: ${text}`);
};
const debug = (text) => {
    console.log(`DEBUG: ${text}`);
};

module.exports = {
    info,
    debug
};
```

Let’s visualize this for a second:



Our `verifyPassword` function is the entry point to the unit of work, and we have a total of two exit points: One that returns a value, and another that calls `logger.info()`.

Unfortunately, we cannot verify that `logger` got called with any traditional means, or without using some jest tricks, and I usually only use those if there’s no other choice, as they tend to make the tests less readable and harder to maintain in many case. (more on that later in this chapter).

Let’s do what we like to do with dependencies: *abstract them*. There are many ways to create a *seam* in our code (remember, seams are places where two pieces of code meet. We can use them to inject fake things*)*. Here’s a list of the most common ways we can use:

Style	Techniques
Standard	Introduce Parameter
Functional	Convert to partial application function Convert to Factory Function
Modular	Abstract Module Dependency
Object Oriented	Inject Untyped Object Inject Interface

4.3 Standard Style: Introduce Parameter Refactoring

The most obvious way we can start this journey is by introducing a new parameter into our code under test. Listing 4.2 shows this:

Listing 4.2 Mock Logger Parameter Injection

```
File: ch4/00-function-param-injection/00-password-verifier00.js
```

```
const verifyPassword2 = (input, rules, logger) => {
    const failed = rules
        .map(rule => rule(input))
        .filter(result => result === false);

    if (failed.length === 0) {
        logger.info('PASSED');
        return true;
    }
    logger.info('FAIL');
    return false;
};
```

And here’s how we could write the simplest of tests for this using a simple closure mechanism:

Listing 4.3: Hand Written Mock Object

```
File: ch4/00-function-param-injection/01-password-verifier00.spec.js

const { verifyPassword2 } = require('./00-password-verifier00');

describe('password verifier with logger', () => {
    describe('when all rules pass', () => {
        it('calls the logger with PASSED', () => {
            let written = '';
            const mockLog = {
                info: (text) => {
                    written = text;
                }
            };

            verifyPassword2('anything', [], mockLog);

            expect(written).toMatch(/PASSED/);
        });
    });
});

```

Notice first that we are naming the variable as `‘mock’XXX (i.e mockLog)` to denote the fact that we have a mock function or object in the test. I use this naming convention because *I* want you, as a *reader of the test,* to know that you should expect an assert (also known as *verification)* against that mock at the end of the test. This sort of naming removes the element of surprise for the reader and makes the test much more predictable. So only use this naming convention for things that are actually mocks.

Here’s our first ever mock object, separated, for focus:

```
let written = '';
const mockLog = {
    info: (text) => {
        written = text;
    }
};
```

It only has one function that mimics the same signature of the logger's `info()` function. It then "saves" the parameter being passed to it (text) so that we can assert that it was called later in the test.

If the `written` variable has the correct test, this proves that our function was called, which means that we have proven the exit point is invoked correctly from our unit of work.

On the `verifyPassword2` side, the refactoring we did is pretty common. and pretty much the same as we did in the previous chapter where we extracted a *stub* as a dependency.

Stubs and mocks are often treated the same in terms of refactoring and introducing seams in our application's code.

What did this simple refactoring into a parameter provide for us?

1. We do not need to explicitly require the `logger` in our code under test anymore. That means that if we ever change the real dependency of the logger, the file under test will have one less reason to change.
2. We now have the ability to inject *any logger* of our choosing into the code under test, as long as it lives up to the same interface (or at least has the `info()` method). In our tests, that's exactly what we're doing. And of course, this means that we can provide a *mock logger* that does our bidding for us: the *mock logger* helps us verify that it was called correctly.

As an aside, the fact that our mock object only mimics a part of the `logger`'s interface (it's missing the `debug()` function) is a form of "duck typing". I spoke more about this idea in chapter 3: "if it walks like a duck, and it talks like a duck, then we can use it is a fake object."

4.4 The Importance of Mock and Stub Differentiation

Why do I care so much what we name each thing? If we can't tell the different between mocks and stubs, or we don't name them correctly, we can end up with tests that are testing multiple things, are less readable and harder to maintain.

Naming things correctly helps us avoid these pitfalls.

Given that a mock represents a requirement from our unit of work ("it calls the logger", "it sends an email" etc..) and that a stub represents incoming information or behavior ("the database query returns false", "the configuration throws an error"), we can set a simple rule of thumb:

It should be OK to have multiple stubs in a test, but you don't usually want to have more than a single mock per test – because that means you're testing more than requirements in a single test.

If we can't (or won't) differentiate between things (naming is key to that) we can end up with multiple mocks per test or asserting our stubs. Both of which can have negative effects on our tests:

1. **Readability**. Your test name will become much more generic in nature and hard to understand. You want people to be able to read the name of the test and know everything that happens or is tested inside of it without needing to read the test's code.

2. **Maintainability**: You could, without noticing or even caring, assert against stubs If you don't differentiate between mocks and stubs. This produces little value to you and increases the coupling between your tests and internal production code. Asserting that you queried a database is a good example for this. It would be much better to test that given that a query to a stub database returns some value, then our application behavior changes or returns a special value.
3. **Trust**: If you have multiple mocks (requirements) in a single test, and the first mock verification fails the test, in 99% of test frameworks, the rest of the test (below the failing assert line) won't even execute because an exception is thrown. This means that the other mocks aren't verified, and we don't get the results from them.

To drive the last point home, imagine a doctor that only sees 30% of their patient's symptoms, but still needs to make a decision. They might make the wrong decision on treatment. If you can't see where all the bugs are or that two things are failing instead of just one (because one of them is "hidden" after the first failure), you're more likely to fix the wrong thing, or fix the wrong place.

The XUnit Patterns book calls this situation "Assertion Roulette" (http://xunitpatterns.com/Assertion%20Roulette.html). I like this name. It's quite a gamble. You start commenting out lines of code in your test and lots of fun ensues (and possibly alcohol).

NOT EVERYTHING IS A "MOCK" Given these reasons, it's unfortunate that people still tend to use the word "mock" for anything that isn't real: "mock database", "mock service". 90% of the time they really mean they are using a stub. It's hard to blame them, though. Frameworks like Mockito, jmock, and most isolation frameworks out there (yeah, I don't call them mocking frameworks, for the same reasons I'm discussing right now), use the word "mock" to denote both mocks and stubs.

There are newer frameworks such as `Sinon` and `testdouble` in JavaScript, `NSubstitute` and `FakeItEasy` in .NET and others that have helped start a change in the naming conventions. I hope this persists.

4.5 Modular Style Mocks

I covered modular dependency injection in the previous chapter, but now we're going to take a look at how we can use it to inject mock objects and answer on them.

4.5.1 Example of production code

Let's take a look at a slightly more complicated example. In this scenario our `verifyPassword` function depends on two external dependencies:

- A `logger`
- And a `configuration service`.

The configuration service provides the `logging level` that is required. Unusually this type of code would be moved into a special logger module but for the purposes of our book's samples I'm putting the logic that calls `logger.info` or `logger.debug` directly in the code under test:

Listing 4.5 A Hard Modular Dependency

```
File: ch4/01-modular-injection/password-verifier-before.js

01    const { info, debug } = require("./complicated-logger");
02    const { getLogLevel } = require("./configuration-service");
03
04    const log = (text) => {
05      if (getLogLevel() === "info") {
06        info(text);
07      }
08      if (getLogLevel() === "debug") {
09        debug(text);
10      }
11    };
12
13    const verifyPassword = (input, rules) => {
14      const failed = rules
15        .map((rule) => rule(input))
16        .filter((result) => result === false);
17
18      if (failed.length === 0) {
19        log("PASSED");
20        return true;
21      }
22      log("FAIL");
23      return false;
24    };
25
26    module.exports = {
27      verifyPassword,
28    };
29
```

Let's assume that we realized we have a bug when we call the `logger` in lines 10 and 13 of this code. We're missing an exclamation mark when calling the logger with a `PASSED` result. How can we prove that this bug exists, or that we fixed it, with a unit test?

Our problem here is that we are importing (or requiring) the modules directly in our code. If we want to replace the logger module, we have to either replace the file or do some other dark magic through jest's API. I wouldn't recommend that usually, because using these techniques leads to more pain and suffering than the usual amount when dealing with code.

4.5.2 Refactoring the production code in a modular injection style

We can abstract away the module dependencies into their own object and allow the user of our module to replace that object, like the code in the following example:

Listing 4.6 Refactoring to a Modular Injection Pattern

```
File: ch4/01-modular-injection/password-verifier-after.js
```

```
01: const originalDependencies = {
02:     log: require('./complicated-logger'),
03: };
04:
05: let dependencies = { ...originalDependencies };
06:
07: const resetDependencies = () => {
08:     dependencies = { ...originalDependencies };
09: };
10:
11: const injectDependencies = (fakes) => {
12:     Object.assign(dependencies, fakes);
13: };
14:
15:
16: const verifyPassword = (input, rules) => {
17:     const failed = rules
18:         .map(rule => rule(input))
19:         .filter(result => result === false);
20:
21:     if (failed.length === 0) {
22:         dependencies.log.info('PASSED');
23:         return true;
24:     }
25:     dependencies.log.info('FAIL');
26:     return false;
27: };
28:
29: module.exports = {
30:     verifyPassword,
31:     injectDependencies,
32:     resetDependencies
33: };
```

There's more production here, and it seems more complex, but this allows us to replace dependencies in our tests in a relatively easy manner if we are somehow force to work in such a modular fasion.

Lines 1-3 will always hold the original dependencies, so that we never lose them between tests.

Line 5 is our layer of indirection. It defaults to the original dependencies, but our tests can direct the code under test to replace that variable with custom dependencies (without knowing anything about the internals of the module) .

Lines 7-13 are the public api that the module exposes to be able to override and reset the dependencies.

Lines 30-32 expose the api to the users of the module.

4.5.3 A test example with modular style injection

Here's what a test for this might look like:

Listing 4.7 testing with modular injection

```
File: ch4/01-modular-injection/password-verifier-injectable.spec.js
```

```
const {
  verifyPassword,
  injectDependencies,
  resetDependencies,
} = require("./password-verifier-injectable");


describe("password verifier", () => {
  afterEach(resetDependencies);

  describe("given logger and passing scenario", () => {
    it("calls the logger with PASS", () => {
      let logged = "";
      const mockLog = { info: (text) => (logged = text) };
      injectDependencies({ log: mockLog });

      verifyPassword("anything", []);

      expect(logged).toMatch(/PASSED/);
    });
  });
});

```

As long as we don't forget to use the `resetDependencies` function after each test, we can now inject modules pretty easily for test purposes. The obvious main caveat is that this approach requires each module to expose `inject` and `reset` functions that can be used from the outside. This might or might not work with your current design limitations, but if it does it's possible to abstract them both into reusable functions and save yourself a lot of boilerplate code.

4.6 Mocks in a Functional Style

Let's jump into a few of the functional styles we can use to inject mocks into our code under test.

4.6.1 Working with a currying Style

Let's implement the currying technique introduced in the previous chapter to perform a more functional style "injection" of our `logger`. We're going to use `lodash` to get currying working without too much boilerplate code:

Listing 4.8: Applying Currying to our function

```
File: ch4/00-function-param-injection/00-password-verifier00.js

const verifyPassword3 = _.curry((rules, logger, input) => {
    const failed = rules
        .map(rule => rule(input))
        .filter(result => result === false);

    if (failed.length === 0) {
        logger.info('PASSED');
        return true;
09:     }
10:     logger.info('FAIL');
11:     return false;
12: });
```

The only change is the call to `_.curry` on line 1, and closing it off at line 12.

And here's a potential test for this type of code might look:

Listing 4.9: Testing a curried function with dependency injection

```
File: /00-function-param-injection/02-password-verifier00.currying.spec.js
const { verifyPassword3 } = require("./00-password-verifier00");

describe("password verifier", () => {
  describe("given logger and passing scnario", () => {
    it("calls the logger with PASS", () => {
      let logged = "";
      const mockLog = { info: (text) => (logged = text) };
      const injectedVerify = verifyPassword3([], mockLog);

      // this partially applied function can be passed arround
      // to other places in the code
      // without needing to inject the logger
      injectedVerify("anything");

      expect(logged).toMatch(/PASSED/);
    });
  });
});
```

Our test is invoking the function with the first two arguments (injecting the `rules` and `logger` dependencies, effectively returning a partially applied function), and then invokes the returned function `injectedVerify()` with the final input, thus showing the reader two things:

1. How this function is meant to be used in real life
2. Showing what the dependencies are in an explicit way

Other than that, it's pretty much the same as in the previous test.

4.6.2 Working with higher order functions and not currying

Here's another variation on this type of design. We're using a higher order function, but without currying. You can tell the following code does not contain currying because we always need to send in all of the parameters as arguments to the function for it to be able to work correctly.

Listing 4.10: Inject a Mock in a Higher Order Function

```
File: ch4/02-function-higher-order-functions/00-password-verifier00.js

const makeVerifier = (rules, logger) => {
    return (input) => {
        const failed = rules
            .map(rule => rule(input))
            .filter(result => result === false);
06:
08:         if (failed.length === 0) {
09:             logger.info('PASSED');
10:             return true;
11:         }
12:         logger.info('FAIL');
13:         return false;
14:     };
15: };
16:
17: module.exports = {
18:     makeVerifier
19: };
20:
```

This time I'm explicitly making a factory function that returns a *preconfigured verifier function (in line 2)*, that already contains the rules and logger in its closure's dependencies (line 1).

Now let's look at what the tests for this look like. The test needs to first call the factory function (line 8, and then call the function that is returned by that function (line 10):

Listing 4.11: testing using a factory function

```
File: ch4/02-function-higher-order-functions/01-password-verifier00.spec.js
1       const { makeVerifier } = require("./00-password-verifier00");
2
3       describe("higher order factory functions", () => {
4         describe("password verifier", () => {
5           test("given logger and passing scenario", () => {
6             let logged = "";
7             const mockLog = { info: (text) => (logged = text) };
8             const passVerify = makeVerifier([], mockLog);
9
10            passVerify("any input");
11
12            expect(logged).toMatch(/PASSED/);
13          });
14        });
15      });
16
```

4.7 Mocks in an Object-Oriented Style

Now that we've covered some functional and modular styles, let's look at the object-oriented styles. People coming from a more object-oriented background would feel much more comfortable with this type of approach. People coming from a more functional background will hate it. But life is about accepting people's differences. So there.

4.7.1 Refactoring Production Code for Injection

If we're moving into a class-based design in JavaScript, here's how this type of injection might look like. Classes have constructors, and we use the constructor to "force" the caller of the class to provide parameters. This is not the only way but, in an object-oriented based design, it is very common and useful because it makes the requirement of those parameters explicit and practically undeniable in strongly typed languages such as Java or C, and when using TypeScript. We want to make sure whoever uses our code knows what is expected to configure it properly.

Listing 4.13 Class based constructor injection

```
File: ch4-v2/04-class-constuctor-injection-duck-typing/00-password-verifier00.js
class PasswordVerifier {
  _rules;
  _logger;

  constructor(rules, logger) {
    this._rules = rules;
    this._logger = logger;
  }

  verify(input) {
    const failed = this._rules
        .map(rule => rule(input))
        .filter(result => result === false);

    if (failed.length === 0) {
      this._logger.info('PASSED');
      return true;
    }
    this._logger.info('FAIL');
    return false;
  }
}

module.exports = {
  PasswordVerifier
};
```

This is just a standard class that takes a couple of constructor parameters, and then uses them inside the `verify` function.

And here’s what a possible test might look like.

Listing 4.14 Injecting a mock logger as a constructor parameter

```
File: /04-class-constuctor-injection-duck-typing/01-password-verifier00.spec.js

const { PasswordVerifier } = require("./00-password-verifier00");

describe("duck typing with function constructor injection", () => {
  describe("password verifier", () => {
    test("logger&passing scenario,calls logger with PASSED", () => {
      let logged = "";
      const mockLog = { info: (text) => (logged = text) };
      const verifier = new PasswordVerifier([], mockLog);
      verifier.verify("any input");

      expect(logged).toMatch(/PASSED/);
    });
  });
  });

```

Mock injection is straightforward. Much like with stubs as we saw in the previous chapter.

If we were to instead use properties rather than a constructor, it would mean that the dependencies are *optional*. With a constructor we're explicitly saying they're not optional.

In strongly types languages like Java or C# it's common to extract the fake logger as a separate class like so:

```
class FakeLogger {
  logged = "";

  info(text) {
    this.logged = text;
  }
}
```

We simply implement the info function in the class, but instead of logging anything, we just save the value being sent as a parameter to the function in a publicly visible variable that we can assert again later in our test.

Also, notice I didn't call the fake object `MockLogger` or `StubLogger`, but **`Fake`**`Logger`. This is because I can reuse this class in multiple different tests. In some tests it might be used as a stub, and in others that fake might be used as a mock object. I use the word "Fake" to denote anything that isn't *real*. Another common word for this sort of thing is "Test Double". Fake is shorter so I like it.

In our tests, we'll instantiate the class and send it over as a constructor parameter, then assert on the `logged` variable of the class like so:

```
test("logger + passing scenario, calls logger with PASSED", () => {
   let logged = "";
   const mockLog = new FakeLogger();
   const verifier = new PasswordVerifier([], mockLog);
   verifier.verify("any input");

   expect(mockLog.logged).toMatch(/PASSED/);
 });
```

4.7.2 Refactoring Production with Interface Injection

Interface play a large role in many object-oriented programs. They are one variation on the idea of polymorphism: allowing one or more objects to be replaceable with one another as long they implement the same interface. In JavaScript and other languages like Ruby, Interfaces are not needed, since the language allows the idea of "duck typing" without needing to cast an object to a specific interface. I won't tough here on the pros and cons of duck typing. You should be able to use either technique as you see fit in the language of your choice.

In JavaScript we can turn to TypeScript to use interfaces. The compiler or transpiler we'll use can help ensure that you we are using types based on their signatures correctly.

In listing 4.15 we have three code files: The first one describes a new `ILogger` interface. The second one describes a `SimpleLogger` that implements that interface, and the third one is our PasswordVerifier that uses only the ILogger interface to get a logger instance, and has no knowledge of the actual type of logger being injected.

Listing 4.15 – Production code gets an ILogger interface

```
// ch4/05-class-constructor-interface-injection/interfaces/logger.ts

export interface ILogger {
    info(text: string);
}

File: ch4/05-class-constructor-interface-injection/simple-logger.ts

import {ILogger} from "./interfaces/logger";

//this class might have dependencies on files or network
class SimpleLogger implements ILogger{
    info(text: string) {
    }
}


File: ch4/05-class-constructor-interface-injection/00-password-verifier.ts

import { ILogger } from "./interfaces/logger";

export class PasswordVerifier {
    private _rules: any[];
    private _logger: ILogger;

    constructor(rules: any[], logger: ILogger) {
        this._rules = rules;
        this._logger = logger;
    }

    verify(input: string): boolean {
        const failed = this._rules
            .map(rule => rule(input))
            .filter(result => result === false);

        if (failed.length === 0) {
            this._logger.info('PASSED');
            return true;
        }
        this._logger.info('FAIL');
        return false;
    }
}
```

Notice that the only things changed in the production code are:

1. I've added a new interface that will be part of production code. I'm changing the design to make the logger replaceable.
2. I'm made the existing logger implement this interface.

3. The type of parameter expected in the class' constructor is that of the interface. This allows me to replace the instance of the logger class with a fake one, instead of having a hard dependency on the real logger.

Here's what a test might look like in a strongly typed language, but with a handwritten fake object that implements the `ILogger` interface:

Listing 4.16: Injecting a hand written mock ILogger

```
//05-class-constructor-interface-injection/01-password-verifier.handwritten.spec.ts

import { PasswordVerifier } from "./00-password-verifier";
import { ILogger } from "./interfaces/logger";

class FakeLogger implements ILogger {
    written: string;
    info(text: string) {
        this.written = text;
    }
}

describe('password verifier with interfaces', () => {
    test('verify, with logger, calls logger', () => {
        const mockLog = new FakeLogger();
        const verifier = new PasswordVerifier([], mockLog);

        verifier.verify('anything');

        expect(mockLog.written).toMatch(/PASS/);
    });
});

```

In this example, I've created a handwritten class called `FakeLogger`. All it does is override the one method in the `ILogger` interface and save the `text` parameter for future assertion. We then "expose" this value as a field in the class called `written`. Once this value is exposed, we can verify that the fake logger was called by checking that field.

I've done this manually because I wanted to you to see that even in Object-Oriented land, the patterns repeat themselves. Instead of having a mock *function*, we now have a mock *object*, but the code and test essentially work just like the previous examples.

NOTE ABOUT INTERFACE NAMING CONVENTIONS: I'm using the naming convention of prefixing the logger interface with an "I" because it's going to be used for polymorphic reasons (i.e., I'm using it to abstract a role in the system). This is not always the case for interface naming in typescript, for example, when we might use interfaces to define the structure of a set of parameters (basically using them as strongly typed structures). In that case naming without an "I" makes sense to me.

For now, think of it like this: If you're going to implement it more than once, you should prefix it with an "I" to make the expected usage of the interface more explicit.

4.8 Dealing with complicated interfaces

But what happens when the interface is more complicated, i.e has more than one or two functions in it, or more than one or two parameters in each function?

4.8.1 Example of a Complicated Interface

Here's an example of such an interface, and the production code verifier that uses the complicated logger, injected as an interface. The `IComplicatedLogger` interface has four functions, each with 1 or more parameters. Every function would require us to "fake" it in our tests, and that can lead to complexity and maintainability problems in our code and our tests.

Listing 4.18 – Working with a more complicated interface (production code)

```
//05-class-constructor-interface-injection/interfaces/complicated-logger.ts

export interface IComplicatedLogger {
    info(text: string)
    debug(text: string, obj: any)
    warn(text: string)
    error(text: string, location: string, stacktrace: string)
}

//: ch4/05-class-constructor-interface-injection/00-password-verifier2.ts

import { IComplicatedLogger } from "./interfaces/complicated-logger";

export class PasswordVerifier2 {
    private _rules: any[];
    private _logger: IComplicatedLogger;

    constructor(rules: any[], logger: IComplicatedLogger) {
        this._rules = rules;
        this._logger = logger;
    }
...
}
```

Notice that the only things changed in the production code are:

1. I've added a new `IComplicatedLogger` interface that will be part of production code. I'm changing the design to make the `logger` replaceable.
2. I'm leaving off the implementation of a real logger, because it's not relevant for our examples. That's also the nice thing about abstracting away things with interface: we can stay away from referencing them directly.
3. The type of parameter expected in the class' constructor is that of the `IComplicatedLogger` interface. This allows me to replace the instance of the logger class with a fake one, just like we did before.

4.8.2 Writing tests with complicated interfaces

Here's what the test looks like. It has to override each and every interface function, which creates long and annoyingboilerplate code:

Listing 4.19: Test code with a complicated logger interface

```
File: ch4/05-class-constructor-interface-injection/03-password-
       verifier.jestFn.longinterfaces.spec.ts

import { IComplicatedLogger } from "./interfaces/complicated-logger";
import { PasswordVerifier2 } from "./00-password-verifier2";

describe("working with long interfaces", () => {
  describe("password verifier", () => {
    class FakeComplicatedLogger implements IComplicatedLogger {
      infoWritten = "";
      debugWritten = "";
      errorWritten = "";
      warnWritten = "";

      debug(text: string, obj: any) {
        this.debugWritten = text;
      }

      error(text: string, location: string, stacktrace: string) {
        this.errorWritten = text;
      }

      info(text: string) {
        this.infoWritten = text;
      }

      warn(text: string) {
        this.warnWritten = text;
      }
    }
...

...
    test("verify passing, with logger, calls logger with PASS", () => {
      const mockLog = new FakeComplicatedLogger();

      const verifier = new PasswordVerifier2([], mockLog);
      verifier.verify("anything");

      expect(mockLog.infoWritten).toMatch(/PASSED/);
    });

    test("A more JS oriented variation on this test", () => {
      const mockLog = {} as IComplicatedLogger;
      let logged = "";
      mockLog.info = (text) => (logged = text);

      const verifier = new PasswordVerifier2([], mockLog);
      verifier.verify("anything");

      expect(logged).toMatch(/PASSED/);
    });
  });
});
```

In line 6, we're declaring, again, a `FakeLogger` class, that implements the `IComplicatedLogger` interface. Look at how much boilerplate code we have here. This will be especially true for those of us working in strongly typed object-oriented languages such as Java, C# and C++. There are ways around all this boilerplate code which we will touch on in the next chapter.

4.8.3 Downsides of using Complicated Interfaces directly

There are other downsides to using long, complicated interfaces in our tests.

1. If we're saving arguments being sent in manually, it's more cumbersome to verify multiple arguments across multiple methods and calls.
2. It's likely that we are depending on 3rd party interfaces instead of internal ones, and this will end up making our tests more brittle as time goes by.
3. Even if we are depending on internal interfaces, long interfaces have more reasons to change, and now, so do our tests.

So, what does this mean for us?

I highly recommend to use only fake interfaces that answer both of these conditions:

1. You control those interfaces (they are not made by a 3rd party).
2. They are adapted to the needs of your unit of work or component.

4.8.4 The Interface Segregation Principle

The 2nd point might need a bit of explanation. It relates to the *interface segregation principle* (ISP for short), see https://en.wikipedia.org/wiki/Interface_segregation_principle for more info. *ISP* means that if you have an interface that contains more functionality than you require, create a small, simpler, adapter interface that contains just the functionality you need, preferably with less functions, better names and less parameters.

This will end up making our tests much simpler. By abstracting away our real dependencies, when the real complicated interfaces change, we won't need to change our tests, only a single adapter class file somewhere. We'll see an example of this in the next chapter.

4.9 Partial Mocks

It's possible, both in JavaScript and in a most other languages and associated test frameworks, to "take over" existing objects and functions and "spy" on them. By spying on them we can later check if they were called, how many times and with which arguments.

This essentially can turn *parts* of real objects into mock functions, while keeping the rest of the object as a real object. This can create more complicated tests that are more brittle, but can sometimes be a viable option, especially if you're dealing with legacy code (see chapter 10 for more).

4.9.1 A Functional Partial Mock Example

Here’s an example of what such a test might look like. We create the real logger, and then simply override one of its existing real functions using a custom function.

Listing 4.20: A Partial Mock Example

```
// ch4/06-partial-mock/02-password-verifier.partial.spec.ts

1     import { PasswordVerifier } from "./00-password-verifier";
2     import { RealLogger } from "./real-logger";
3
4     describe("password verifier with interfaces", () => {
5       test("verify, with logger, calls logger", () => {
6         const testableLog: RealLogger = new RealLogger();
7         let logged = "";
8         testableLog.info = (text) => (logged = text);
9
10        const verifier = new PasswordVerifier([], testableLog);
11        verifier.verify("any input");
12
13        expect(logged).toMatch(/PASSED/);
14      });
15    });
16
```

In the first test, I’m instantiating a `RealLogger()` and then in the next line I’m replacing one of its existing functions to be a fake one. More specifically, a mock function, that allows me to track its latest invocation parameter using a custom variable.

But the important part here is that in line `8`, the `testableLog` variable is a *partial mock*. That means that at least some of its internal implementation is not fake and might have real dependencies and logic in it.

Sometimes it makes sense to use partial mocks, especially when you’re working with legacy code, and you might need to isolate some existing code from its dependencies. I’ll touch more on that in the chapter on legacy code later in the book.

4.9.2 An Object Oriented Partial Mock Example

One object oriented version of this uses inheritance to override functions from real classes so that we can verify they were call. Here’s how we do the same using inheritance and overrides in JS:

Listing 4.21: A Object Oriented Partial Mock Example

```
// ch4/06-partial-mock/02-password-verifier.partial.spec.ts

class TestableLogger extends RealLogger {
  logged = "";
  info(text) {
    this.logged = text;
  }
  //the error() and debug() functions
  // are still "real"
}
describe("partial mock with inhertiance", () => {
  test("verify with logger, calls logger", () => {
    const mockLog: TestableLogger = new TestableLogger();

    const verifier = new PasswordVerifier([], mockLog);
    verifier.verify("any input");

    expect(mockLog.logged).toMatch(/PASSED/);
  });
});
```

I inherit from the real logger class in my tests, and then using the inherited class, and NOT the original class, in my tests. This technique is commonly called "Extract & Override" and you can find this and more by looking into Michael Feathers' book "Working Effectively with Legacy Code". I'll touch on this and other techniques later in the book.

A few things to note about this example:

- I'm naming the fake logger class "TestableXXX" because it is a testable version of real production code – containing a mix of fake and real code, and this convention helps me make this explicit for the reader.
- I'm putting the class right along side my tests. My production code doesn't need to know that this class exists.
- This "Extract & Override" style requires that my class in production code allows inheritance and that the function allows overriding. In JS this is less of an issue but in Java and C# these are explicit design choices that need to be made (usually. There are frameworks that allow circumventing this rule and we'll discuss them in the next chapter)
- In this scenario we're inheriting from a class that we're not testing directly (`RealLogger`) so that we can test another class (`PasswordVerifier`). However, this technique can be used quite effectively to isolate and "stub" or "mock" single functions from classes that you're directly testing, and we'll touch more on that later in the book when we talk about legacy code and refactoring techniques.

4.10 What about Spies?

Readers with some background in using various test frameworks in JS may have noticed that I've not included the word "Spy" in any of the examples here. Spy was originally designed as the idea that you can look at a *real* function and "spy" on it without altering its behavior.

You could later assert that the function got called with the right inputs or that it returned the correct outputs. That usage is relatively neglected, and I haven't seen that pattern in the wild almost anywhere.

In some JS frameworks, the word "spy" seems to have been adopted to mean the same thing as a partial mock or a partial stub. In that sense, if you adopt that meaning, both of the examples of partial mocks I've shown here can also be called "spies". Do note that outside of JS I don't real seeing any usage of the wold "Spy" in various test frameworks so my recommendation: whenever you see someone mention "spy" think "partial mock/stub" because chance are that's what they're trying to achieve.

When in doubt, think of "Spy" as "Partial Mock/Stub"

4.11 Summary

We've covered a lot of ground in this chapter:

- The reasoning for interaction testing
- Refactoring in various styles (both function, modular and object oriented
- How to deal with complicated interfaces
- Partial mocks and spies

In the next chapter we'll dive into the world of Isolation frameworks and see what they bring to the table.

5 Isolation (mocking) frameworks

This chapter covers

- **Defining isolation frameworks and how they help**
- **Two main “flavors” of frameworks**
- **Faking modules with jest**
- **Faking functions with jest**
- **Object Oriented Fakes with Substitute**

In the previous chapters, we looked at writing mocks and stubs manually and saw the challenges involved, especially when the interface we’d like to fake requires us to create long manual and error prone repetitive code.

We kept having to declare custom variables, create custom functions or inherit from classes that use those variables and basically make things a bit more complicated that they need to be (most of the time).

In this chapter, we’ll look at some elegant solutions for these problems in the form of an *isolation framework*—a reusable library that can create and configure fake objects *at runtime*. These objects are referred to as *dynamic stubs* and *dynamic mocks*.

I call them isolation frameworks because they allow you to isolate the unit of work from its dependencies. You’ll find that many resources will refer to them as “mocking frameworks”. I *try to avoid calling them mocking frameworks because they can be used for both mocks and stubs.* We’ll take a closer look at a few of the JavaScript frameworks available and how we can use them in modular, functional and object-oriented designs. You’ll see how you can use such a framework to test various things and to create stubs, mocks, and other interesting things.

But the actual frameworks I’ll present here a aren’t the point. While using them, you’ll see the specific values that their API promotes in your tests (readability, maintainability, robust long-lasting tests, and more) and find out what makes an isolation framework good and, alternatively, what can make it a drawback for your tests.

5.1 Defining Isolation Frameworks

I'll start with a basic definition that may sound a bit bland, but it needs to be generic in order to include the various isolation frameworks out there.

> **DEFINITION** **An *isolation framework* is a set of programmable APIs that allow the dynamic creation, configuration and verification of mocks and stubs, either in object or function form; Using a framework, these tasks can often be simpler, faster, and shorter than hand-coding them.**

Isolation frameworks, when designed well, can save the developer from the need to write repetitive code to assert or simulate object interactions, and if applied in the right places, they can help make tests last many years without making the developer come back to fix them on every little production code change. If they're applied badly they can cause confusion and full on abuse of these frameworks, to the point where we either can't read or can't trust our own tests – so be wary. I'll discuss some Dos and DONTs later on in the part "The Test Code".

5.2 A quick look at the Isolation frameworks landscape

Isolation frameworks exist for most languages that have a unit testing framework associated with them. Some examples:

- C++ has mockpp and other frameworks
- Java has Mockito, PowerMock and JMock among others.
- C# has several well-known ones including Moq, FakeItEasy and NSubstitute.
- Kotlin has MockK and Mockito-kotlin

Because JavaScript supports multiple paradigms of programming design, we can split the frameworks in our world to two main flavors:

(REVIEWERS: I'm looking for a good name of these categories. Maybe something like "Loose Frameworks" and "Typed Frameworks". Ideas?)

- **Loose JS Isolation Frameworks:**
  - Vanilla JS friendly loose-typed isolation frameworks. These usually also lend themselves better to more functional style code because they require less "ceremony" and boilerplate code to do their work.
- **Typed JS Isolation Frameworks:**
  - More Object Oriented/TypeScript friendly isolation frameworks. Very useful when dealing with whole classes and interfaces.

Some of the more prominent **functional-friendly** isolation frameworks are:

- jest (a test runner and test framework, it also contains APIs to fake various things)
- sinon
- testdouble
- jasmine (another test framework which contains isolation APIs).

jest does support some TypeScript friendly features but is a bit awkward to use when dealing with more object oriented faking that is not module based, as we'll see shortly.

In the **Type-Friendly** realm, you can find some type-friendly frameworks that are often geared more towards both strong typing and as well as an object-oriented mindset:

- ts-mockito (based on the design of Java's Mockito library)
- typemoq (based on C#'s Moq Library)
- substitute.js (Based on C#'s NSubstitute)
- ts-mockery
- OmniMock
- And more..

5.3 Choosing a loose vs typed "flavor"

Which one you end up choosing to use in your project will depends on a few things like taste, style and readability, but the main question to start with should be:

What type of dependencies will you need to fake mostly?

- **Module** dependencies (imports, requires):
  - jest and other loosely typed frameworks should work well
- **Functional** (single and higher order functions, simple parameters and values)
  - jest and other loosely typed frameworks should work well
- **Full objects**, object hierarchies and interfaces:
  - Look into the more object-oriented frameworks such as substitute.js.

Let's go back to our password verifier and see how we can fake the same types of depednencies we did in previous chapters, but this time, using a framework.

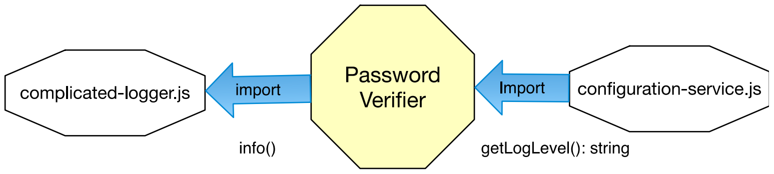
5.4 Faking Modules Dynamically

For people who are trying to test code with direct dependencies on modules using `require` or `import`, Isolation frameworks such as jest or sinon present a powerful ability: The ability to fake an entire module dynamically, with very little code involved.

Figure 5.1 presents an a password verifier with two dependencies:

- A configuration service that helps decide what is the logging level
- (INFO or ERROR)
- A logging service that we call as the exit point of our unit of work, whenever we verify a

password:



The arrows represent the flow of behavior through the unit of work. Another way to think about the arrorws is through the terms *command* and *query*. We are querying the configuration service (to get the log level), but we are sending commands to the logger (to log).

NOTE ABOUT COMMAND/QUERY **There is a school of design that falls under the ideas of Command/Query separation. If you'd like to learn more about these terms, I highly recommend reading about this at https://martinfowler.com/bliki/CommandQuerySeparation.html . It is very beneficial as you navigate your way around different design ideas – but we're not touching on this too much in this book)**

First, here's a password verifier that has a hard dependency on a logger module.

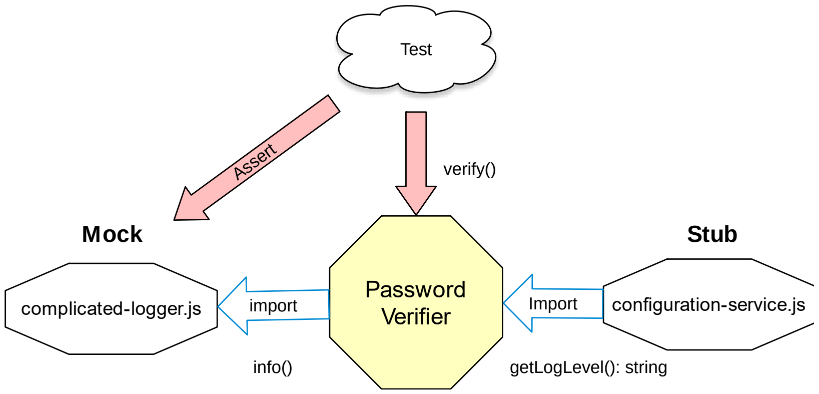
5.1 Code with hard coded modular dependencies

```
File: ch5/00-modular-faking/password-verifier.js

const { info, debug } = require("./complicated-logger");
const { getLogLevel } = require("./configuration-service");

const log = (text) => {
  if (getLogLevel() === "info") {
    info(text);
  }
  if (getLogLevel() === "debug") {
    debug(text);
  }
};

const verifyPassword = (input, rules) => {
  const failed = rules
    .map((rule) => rule(input))
    .filter((result) => result === false);

  if (failed.length === 0) {
    log("PASSED");
    return true;
  }
  log("FAIL");
  return false;
};

module.exports = {
  verifyPassword,
};
```

In this example we're forced to find a way to do two things:

- Simulate (Stub) values returned from the `configuration` service's `getLogLevel()` function
- Verify (Mock) the `logger` module's `info()` function was called /not called

Here's a more visual representation of this:



We have many options in this situation on how to proceed, but since we started with jest as our test framework, we'll stick to jest for the examples in this chapter.

Jest presents us with a few ways to accomplish this, and one of the cleaner ways it presents is using `jest.mock([module name])` at the top of the spec file, followed by us requiring the fake modules in our tests so that we can configure them:

Listing 5.2: Faking direct modules with jest.mock()

```
File: ch5/00-modular-faking/password-verifier.jestmocks.spec.js

jest.mock("./complicated-logger"); #A
jest.mock("./configuration-service");

const { stringMatching } = expect;
const { verifyPassword } = require("./password-verifier");
const mockLoggerModule = require("./complicated-logger"); #B
const stubConfigModule = require("./configuration-service");

describe("password verifier", () => {
  afterEach(jest.resetAllMocks); #C

  test(`with info log level and no rules,
          it calls the logger with PASSED`, () => {
    stubConfigModule.getLogLevel.mockReturnValue("info"); #D

    verifyPassword("anything", []);

    expect(mockLoggerModule.info)
.toHaveBeenCalledWith(stringMatching(/PASS/)); #E
  });
20
21      test(`with debug log level and no rules,
22              it calls the logger with PASSED`, () => {
23        stubConfigModule.getLogLevel.mockReturnValue("debug"); #F
24
25        verifyPassword("anything", []);
26
27        expect(mockLoggerModule.debug)
.toHaveBeenCalledWith(stringMatching(/PASS/)); #F
28      });
29    });
```

There's quite a bit going on here so let's dissect it.

#A The first two lines in the file are required to contain the module faking API calls.
#B In lines 6-7 we're getting the fake instances of the modules by requiring them in our test, so that we can configure them later on.
#C Line 10: We're telling jest to reset any fake module behavior between tests.
#D Line 14: We're configuring the stub logger module's `getLogLevel()` function to return a fake "info" value.
#E In line 18 we're asserting the mock logger's `info()` function was called correctly using the `stringMatching` helper function that exists on the expect object (we're getting that in line 4)
#F Lines 23 + 27: Reconfiguring the stub config and asserting on the mock logger as done previously.

By using jest here, I've saved myself a bunch of typing and the tests still looks rather readable.

5.4.1 Some things to notice about jest's API:

Jest uses the word "mock" almost everywhere, whether we're stubbing things or mocking them, which can be a bit confusing. It'd be great if it had the word "stub" aliased to "mock" to make things more readable.

Also, due to the way JavaScript "hoisting" works, the lines faking the modules (lines 1,2) will need to be at the top of the file. You can read more about this here: https://medium.com/better-programming/hoisting-in-javascript-6af97650dbb2

Also note that Jest has many other APIs and abilities and its worth exploring them if you're interested in using it. Head over to https://jestjs.io/ to get the full picture. It's beyond the scope of this book, which is mostly about patterns, not tools.

A few other frameworks, among them sinon.js (https://sinonjs.org), also support faking modules. Sinon is quite pleasant to work with, as far as isolation frameworks go, but like many other frameworks in the JS world, and much like jest, it contains *too many ways* of accomplishing the same task, and that can often be confusing. Still, faking modules by hand can be quite annoying without these frameworks at hand.

5.4.2 Consider abstracting away direct dependencies

The **good news** about the `jest.mock()` API and others like it, is that it solves a very real need for developers who are stuck trying to test modules that have baked in dependencies that are not easily changeable (i.e code they **cannot control**). This type of issue is very prevalent in legacy code situations which I discuss in a later chapter).

The **bad news** about this API is that it also solves the same "problem" for code that we **do control**, and that might have benefitted from abstracting away the real dependencies behind simpler shorter internal APIs. This approach, also known as "onion architecture" or "hexagonal architecture" or "ports and adapters" is very useful for the long-term maintainability of our code. You can read more about this type of architecture at https://alistair.cockburn.us/hexagonal-architecture/ .

Why are direct dependencies potentially problematic? By using those APIs directly, we're also forced into faking the direct module APIs in our test instead of their abstractions. we're gluing the design of those direct APIs to the implementation of the tests – which means that if (really, "when") those APIs change, we'll also need to change many of our tests.

Here's a quick example. Imagine depending on a well know JS logging framework (such as Winston) and depending on it directly in hundreds and thousands of places in the code.

Then imagine that Winston has released a breaking upgrade. Lots of pain will ensue, that can be addressed much earlier before things get out of hand.

One simple way to accomplish this would be simple abstraction to a single "adapter" file that is the only one holding the reference to that logger can be made. That abstraction can expose a simpler, internal logging API that we do control and thus are able to prevent large scale breakage across our code.

I'll return to this subject in the chapter "design and testability".

5.5 Functional Mocks and Stubs - Dynamically

We covered modular dependencies, so let's turn to faking simple functions. We've already done that plenty of times in the previous chapters, but we've always done it by hand. That works great for stubs, but for mocks it gets annoying fast.

Here's the manual way we used before:

Listing 5.3: manually mocking a function to very it was called

```
1    test("given logger and passing scenario", () => {
2      let logged = "";
3      const mockLog = { info: (text) => (logged = text) };
4      const passVerify = makeVerifier([], mockLog);
5
6      passVerify("any input");
7
8      expect(logged).toMatch(/PASSED/);
9    });
```

It works – we're able to verify the logger function was called, but:

- In line 2 we have to declare a custom variable to hold the value passed in
- In line 3 we have to generically save the passed in value to that variable
- In line 8 we assert on the value of that variable

That's a lot of work that can become very repetitive.

Enter Isolation frameworks like *jest, sinon* and *testdouble.* In our case: Jest.

Jest.fn() is the simplest way to get rid of such code. Here's how we can use it:

Listing: 5.4: using jest.fn() for simple function mocks

```
File: ch5/01-function-faking/01-password-verifier00.spec.js

1     test('given logger and passing scenario', () => {
2           const mockLog = { info: jest.fn() };
3           const verify = makeVerifier([], mockLog);
4           verify('any input');
5
6           expect(mockLog.info)
7             .toHaveBeenCalledWith(stringMatching(/PASS/));
8         });
```

Compare this code with the previous example. It's subtle but saves plenty of time.

- In line 2 we're using jest.fn() to get back a function that is automatically tracked by jest, so that:
- we can query it later using jest's api in lines 6 and 7.

It's small and cute, and works great any time you need to track calls to a specific function.

The stringMatching function is an example of a "matcher". "*Matcher*" is usually defined as a utility function that can assert on the value of a parameter being sent into a function.

The jest docs are using the term a bit more liberally but you can find the full list of matchers over at https://jestjs.io/docs/en/expect .

To summarize: jest.fn() works well for single-function-based mocks and stubs. Let's move to a more object-oriented challenge.

5.6 Object Oriented Dynamic Mocks and Stubs

As we've just seen, Jest.fn() is an example of a single-function faking utility function. It works well in a functional world, but it breaks down a bit when we try to use it on full blown API interfaces or classes that contain multiple functions.

Here's an example trying to tackle the IComplicatedLogger we've looked at in the previous chapter.

Listing 5.5: the IComplicatedLogger interface

```
File: ch5/02-longinterfaces-faking/interfaces/complicated-logger.ts

export interface IComplicatedLogger {
    info(text: string, method: string)
    debug(text: string, method: string)
    warn(text: string, method: string)
    error(text: string, method: string)
}
```

Creating a handwritten stub or mock for this interface may be very time consuming, because you'd need to remember the parameters on a per-method basis, as this listing shows.

Listing 5.6 hand written stubs create lots of boilerplate code

```
File: ch5/02-longinterfaces-faking/02-password-verifier.manual.spec.ts

import { PasswordVerifier2 } from "./00-password-verifier2";
import { IComplicatedLogger } from "./interfaces/complicated-logger";

describe("working with long interfaces", () => {
  describe("password verifier", () => {
    class FakeLogger implements IComplicatedLogger {
      debugText = "";
      debugMethod = "";
      errorText = "";
      errorMethod = "";
      infoText = "";
      infoMethod = "";
      warnText = "";
      warnMethod = "";

      debug(text: string, method: string) {
        this.debugText = text;
        this.debugMethod = method;
      }

      error(text: string, method: string) {
        this.errorText = text;
        this.errorMethod = method;
      }
      ...
    }


    test("verify,w logger & passing, calls logger with PASS", () => {
      const mockLog = new FakeLogger();
      const verifier = new PasswordVerifier2([], mockLog);

      verifier.verify("anything");

      expect(mockLog.infoText).toMatch(/PASSED/);
    });
  });
});
```

What a mess. Not only is this handwritten fake time consuming and cumbersome to write, what happens if you want it to return a specific value somewhere in the test, or simulate an error from a function call on the logger? We can do it, but the code gets ugly fast.

Using an isolation framework, the code for doing this becomes trivial, readable, and much shorter. Let’s try using `jest.fn()` for the same task and see where we end up:

Listing 5.7: mocking individual interface functions with jest.fn()

```
File: ch5/02-longinterfaces-faking/03-password-verifier.jestFn.longinterfaces.spec.ts

import { IComplicatedLogger } from "./interfaces/complicated-logger";
import { PasswordVerifier2 } from "./00-password-verifier2";
import stringMatching = jasmine.stringMatching;

```

```
5     describe("working with long interfaces", () => {
6       describe("password verifier", () => {
7         test("verify, w logger & passing, calls logger with PASS", () => {
8           const mockLog: IComplicatedLogger = {
9             info: jest.fn(),
10            warn: jest.fn(),
11            debug: jest.fn(),
12            error: jest.fn(),
13          };
14
15          const verifier = new PasswordVerifier2([], mockLog);
16          verifier.verify("anything");
17
18          expect(mockLog.info)
.toHaveBeenCalledWith(stringMatching(/PASS/));
19        });
20      });
21    });
```

Not too shabby. In line 8, We simply outline our own object, and attach a `jest.fn()` function to each of the function sin the interface. This does save a lot of typing **but** has one important caveat:

> **Whenever the interface changes (a function is added for example), we'll have to go back to the code that defines this object and add that method in there.**

(Note: With plain JS this issue would be less of an issue, but can still create some complications if the code under test uses a function we didn't define in the test.)

In any case, it might be wise to push the creation of such a fake object into a factory helper method so that the creation only exists in a single place.

5.6.1 Switching to a type-friendly Isolation Framework

I mentioned before that there are two categories of Isolation frameworks. We were using the first (loosly-typed, function-friendly kind). Let's switch to the second category and try **substitute.js** on for size (we have to choose one, and I like the C# version of this framework a lot, and have used it in the previous edition of this book).

With *Substitute* (and the assumption of working with TypeScript), we can write code like this:

Listing 5.8: using Substitute.js to fake a full interface

```
File: ch5/02-longinterfaces-faking/03-password-verifier.substitute.longinterfaces.spec.ts

01    import { IComplicatedLogger } from "./interfaces/complicated-logger";
02    import { PasswordVerifier2 } from "./00-password-verifier2";
03    import { Substitute, Arg } from "@fluffy-spoon/substitute";
04
05    describe("working with long interfaces", () => {
06      describe("password verifier", () => {
07        test("verify, w logger & passing, calls logger w PASS", () => {
```

```
08          const mockLog = Substitute.for<IComplicatedLogger>(); #A
09
10          const verifier = new PasswordVerifier2([], mockLog);
11          verifier.verify("anything");
12
13          mockLog.received().info(
14            Arg.is((x) => x.includes("PASSED")), #B
15            "verify"
16          );
17        });
18      });
19    });
20
```

#A In line `08` we're using a `Substitute.for(T)` to generate the fake object, which absolves us of caring about any other functions that then one we're testing against, even if the object signature changes in the future.
#B In line 14 we're using `.received()` as our verification mechanism, as well as another argument matcher (`Arg.is`), this time from Substitute's Api, that works just like `stringMatches` from jasmine, but is only the tip of the iceberg

The added benefit here is that if new functions are added to the object's signature, we will be less likely to change the test and there's no need to add those functions in any tests that use the same object signature.

You can find the rest of the documentation for this framework at https://www.npmjs.com/package/@fluffy-spoon/substitute

Arrange-act-assert

Notice how the way you use the isolation framework matches nicely with the structure of arrange-act-assert. You start by arranging a fake object, you act on the thing you're testing, and then you assert on something at the end of the test.

It wasn't always this easy, though.

In the olden days (around 2006) most of the open source isolation frameworks didn't support the idea of arrange-act-assert and instead used a concept called record-replay (we're talking about Java and C#).

Record-replay was a nasty mechanism where you'd have to tell the isolation API that its fake object was in record mode, and then you'd have to call the methods on that object as you expected them to be called from production code.

Then you'd have to tell the isolation API to switch into replay mode, and only *then* could you send your fake object into the heart of your production code.

An example can be seen on the Google testing blog:

https://www.baeldung.com/easymock

Compared to today's abilities to write tests that use the far more readable arrange-act-assert model, this tragedy cost many developers millions of combined hours in painstaking test reading, to figure out exactly where the test failed.

If you have the first edition of this book, you can see an example of record-replay when I showed Rhino Mocks in these chapters (which had the same design initially).

OK, that was mocks. What about stubs?

5.7 Stubbing Behavior Dynamically

Simulating return values with jest, both for modular or functional dependencies has a very simple API: `mockReturnValue()` and `mockReturnValueOnce()`

Listing 5.9: Stubbing a value from a fake function with jest.fn()

```
File: ch5/03-stubs/04-jestFn-mockReturnValue.spec.ts

test("fake same return values", () => {
  const stubFunc = jest.fn()
      .mockReturnValue("abc");

  //value remains the same
  expect(stubFunc()).toBe("abc");
  expect(stubFunc()).toBe("abc");
  expect(stubFunc()).toBe("abc");
});

test("fake multiple return values", () => {
  const stubFunc = jest.fn()
    .mockReturnValueOnce("a")
    .mockReturnValueOnce("b")
    .mockReturnValueOnce("c");

  //value remains the same
  expect(stubFunc()).toBe("a");
  expect(stubFunc()).toBe("b");
  expect(stubFunc()).toBe("c");
  expect(stubFunc()).toBe(undefined);
});
```

Notice that in the first test, we're setting a **permanent** return value for the duration of the test. This is my preferred method of writing tests if I can use it, because it makes the tests simple to read and maintain later on. If we do need to simulate multiple values we can use `mockReturnValueOnce`. When these values are done "replaying" the stub returns an `undefined` value.

If you need to simulate an error or do anything a bit more complicated, you can use `mockImplementation()` and `mockImplemtationOnce()`:

```
yourStub.mockImplementation(() => {
  throw new Error();
});
```

5.7.1 An Object-Oriented example with a mock and a stub

Let's add another ingredient into our password verifier equation.

- Let's say that the password verifier is NOT active during a special maintenance window when software is being updated.
- When a maintenance window is active, calling `verify()` on the verifier will cause it to call `logger.info()` with "under maintenance".
- Otherwise it would call it with a "passed" or "failed" result.

For this purpose (and for the purpose of showing an object oriented design decision) we're introducing a `MaintenanceWindow` interface that will be injected into the constructor of our password verifier. Figure 5.1 illustrates this:

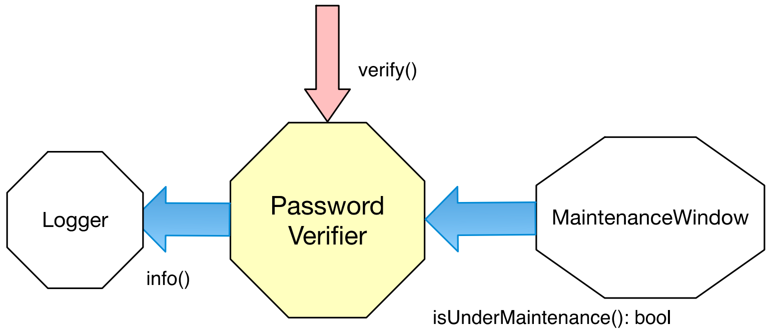


Figure 5.1 using MaintenanceWindow interface.

And here's the code for the new password verifier that uses the new dependency:

Listing 5.10: Password Verifier with a MaintenanceWindow Dependency

File: ch5/03-stubs/00-password-verifier3.ts

```
import { IComplicatedLogger }
from "../02-longinterfaces-faking/interfaces/complicated-logger";
import { MaintenanceWindow } from "./maintenance-window";

export class PasswordVerifier3 {
  private _rules: any[];
  private _logger: IComplicatedLogger;
  private _maintenanceWindow: MaintenanceWindow;

  constructor(
    rules: any[],
    logger: IComplicatedLogger,
    maintenanceWindow: MaintenanceWindow
  ) {
    this._rules = rules;
    this._logger = logger;
    this._maintenanceWindow = maintenanceWindow;
  }

  verify(input: string): boolean {
    if (this._maintenanceWindow.isUnderMaintenance()) {
      this._logger.info("Under Maintenance", "verify");
      return false;
    }
    const failed = this._rules
      .map((rule) => rule(input))
      .filter((result) => result === false);

    if (failed.length === 0) {
     this._logger.info("PASSED", "verify");
     return true;
    }
    this._logger.info("FAIL", "verify");
    return false;
   }
  }

```

The `MaintenanceWindow` interface is injected as a constructor parameter (i.e "constructor injection") and is used to determine where to execute or not execute the password verification and send the proper message to the logger.

5.7.2 Stubs and Mocks with Substitute.JS

This time, we'll use substitute.js to create a stub of the `MaintenanceWindow` interface, and a mock of the `IComplicatedLogger` interface. Figure 5.2 visualizes this:

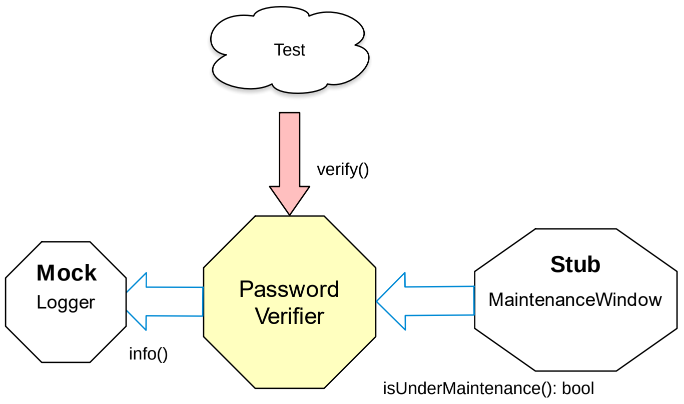


Figure 5.2: A `MaintenanceWindow` **dependency**

Creating stubs or mocks with substitute works the same way. we'll use the `Substitute.for<T>()` function. We can configure stubs with the `.returns()` function, and verify mocks with the `.recieved()` function. Both of these are part of the fake object that is returned from `Substitute.for<T>()`. Here's what are stub creation and configuration would look like:

```
const stubMaintWindow = Substitute.for<MaintenanceWindow>();
stubMaintWindow.isUnderMaintenance().returns(true);
```

and mock creation and verification looks like this:

```
const mockLog = Substitute.for<IComplicatedLogger>();
...
/// later down in the end of the test…
mockLog.received().info("Under Maintenance", "verify");
```

Here’s what the full code for a couple of tests that use a mock nd a stub might look like:

Listing

File: ch5/03-stubs/05-password-verifier3.substitute.spec.ts

```
import { Substitute } from "@fluffy-spoon/substitute";
import { PasswordVerifier3 } from "./00-password-verifier3";
import { IComplicatedLogger }
from "../02-longinterfaces-faking/interfaces/complicated-logger";
import { MaintenanceWindow } from "./maintenance-window";

const makeVerifierWithNoRules = (log, maint) =>
  new PasswordVerifier3([], log, maint);

describe("working with substitute part 2", () => {
  test("verify, with logger, calls logger", () => {
    const stubMaintWindow = Substitute.for<MaintenanceWindow>();
    stubMaintWindow.isUnderMaintenance().returns(true);
    const mockLog = Substitute.for<IComplicatedLogger>();
    const verifier = makeVerifierWithNoRules(mockLog, stubMaintWindow);

    verifier.verify("anything");

    mockLog.received().info("Under Maintenance", "verify");
  });

  test("verify, with logger, calls logger", () => {
    const stubMaintWindow = Substitute.for<MaintenanceWindow>();
    stubMaintWindow.isUnderMaintenance().returns(false);
    const mockLog = Substitute.for<IComplicatedLogger>();
    const verifier = makeVerifierWithNoRules(mockLog, stubMaintWindow);

    verifier.verify("anything");

    mockLog.received().info("PASSED", "verify");
  });
});
```

We can successfully and relatively easily simulate values in our tests with dynamically created objects. I encourage you to look at the other frameworks mentioned he and at the “tools” index in this book to research your favorite flavor of isolation framework you’d like to use. I’ve only used substitute as an example. It’s not the only nice framework out there.

The nice thing about this test is that it requires no handwritten fakes, but notice how it’s already starting to take a toll on the readability for the test reader. Functional designs are usually much slimmer than this. In an object oriented setting, sometimes this is a necessary evil. However, we could easily refactor creation of various helpers, mocks and stubs to helper functions as we refactor our code so the test can become simpler and shorter to read. More on that in part 3: the test code.

5.8 Advantages and traps of isolation frameworks

From what we've covered in this chapter, I can see distinct advantages to using isolation frameworks:

- *Easier modular faking*— module dependencies can be hard to get around without some boilerplate code. This point can also be counted as a negative as explained earlier, because it encourages us to have strongly coupled code to 3rd party implementation.
- *Easier simulation of values or errors*—With manually written mocks, it can be difficult to simulate across a complicated interface. Frameworks help a lot.
- *Easier fakes creation*—Isolation frameworks can be used for creating both mocks and stubs more easily.

Although there are many advantages to using isolation frameworks, there are possible dangers. Here are a few things to watch out for:

5.8.1 You don't need mock objects most of the time

The biggest trap isolation frameworks give you is making it easy to fake anything easily, and trapping you into the thought that you actually need mock objects in the first place.

I'm not saying you won't need stubs, but mock objects shouldn't be the standard operating procedure for most unit tests.

Remember that a unit of work can have three different types of exit points: return values, state change and calling a 3rd party. Only one of these types can benefit from a mock object in your test. The others don't.

I find that in my own tests, mock objects are present in perhaps 2-5% of my tests. The rest of the tests are usually return value or state based tests. For functional designs the amount of mock objects should be nearing zero except some corner cases.

If you find yourself defining a test and verifying an object or function was called, think carefully whether you can prove the same functionality without a mock object, but instead verifying a return value or a change in the behavior of the overall unit of work form the outside (for example, a function throws an exception when it didn't before)

5.8.2 Unreadable test code

Using a mock in a test already makes the test a little less readable, but still readable enough that an outsider can look at it and understand what's going on. Having many mocks, or many expectations, in a single test can ruin the readability of the test so it's hard to maintain or even to understand what's being tested.

If you find that your test becomes unreadable or hard to follow, consider removing some mocks or some mock expectations or separating the test into several smaller tests that are more readable.

5.8.3 Verifying the wrong things

Mock objects allow you to verify that methods were called on your interfaces or that functions were called, but that doesn't necessarily mean that you're testing the right thing.

A lot of people new to tests end up verifying things just because they can, not because it makes sense. Example may include:

- Verifying that an internal function calls another internal function (not an exit point)
- Verifying that a stub was called (incoming dependency should not be verified: it created over specification as we'll discuss later)
- Verifying that something was called simply because someone told you to write a test and you're not sure what should really be tested. (This is a good time to verify that you're understanding the requirements correctly)

5.8.4 Having more than one mock per test

It's considered good practice to test only one concern per test. Testing more than one concern can lead to confusion and problems maintaining the test. Having two mocks in a test is the same as testing several end results of the same unit of work (multiple exit points).

For each exit point, consider writing a separate test, as it could be considered a separate requirement. Chance are that your test names will also become more focused and readable due to only testing one concern. Another way of putting this: If you can't name your test because it does too many things and the name becomes very generic (i.e "XWorksOK") it's time to separate it into more than one test.

5.8.5 Over specifying the tests

If your test has too many expectations (`x.received().X()` and `X.received().Y()` and so on), it may become very fragile, breaking on the slightest of production code changes, even though the overall functionality still works.

Testing interactions is a double-edged sword: test it too much, and you start to lose sight of the big picture—the overall functionality; test it too little, and you'll miss the important interactions between units of work.

Here are some ways to balance this effect:

- *Use stubs instead of mocks when you can.* If you have more than 5% of your tests with mock objects, you might be overdoing it. Stubs can be everywhere. Mocks, not so much. You only need to test one scenario at a time. The more mocks you have, the more verifications will take place at the end of the test, but usually only one will be the important one. The rest will be noise against the current test scenario.
- *Avoid using stubs as mocks if humanly possible.* Use a stub only for faking simulating values into the unit of work under test or to throw exceptions. Don't verify that methods were called on stubs.

5.9 Summary

Isolation frameworks are pretty cool, and you should learn to use them at will. Especially in modular dependency situations they save a lot of time. But it's important to lean toward return-value or state-based testing (as opposed to interaction testing) whenever you can, so that your tests assume as little as possible about internal implementation details. Mocks should be used only when there's no other way to test the implementation, because they eventually lead to tests that are harder to maintain if you're not careful.

Isolation frameworks can help make your testing life much easier and your tests more readable and maintainable. But it's also important to know when they might hinder your development more than they help. In legacy situations, for example, you might want to consider using a different framework based on its abilities. It's all about picking the right tool for the job, so be sure to look at the big picture when considering how to approach a specific problem in testing.

6 Unit Testing Async Code

This chapter covers

- **Async done() and awaits**
- **Integration & Unit Test Levels for Async**
- **Extract Entry Point Pattern**
- **Extract Adapter Pattern**
- **Stubbing, Advancing & Resetting Timers**
- **Observables, Unit and Marble Testing**

When we're dealing with regular synchronous code, waiting for actions to finish is **implicit**. We don't worry about it and we don't really think about it too much.

When dealing with asynchronous code, waiting for actions to finish becomes an **explicit** activity which is under our control. A-synchronicity makes code, and also the tests for that code, potentially more tricky because we have to be explicit about waiting for actions to complete.

Let's start with a simple fetching example to show the issue.

6.1 Dealing with Async Data Fetching

Let's say we have some a module that checks whether our website at example.com is alive. It does this by fetching the context form the main url and checking for a specific word: "illustrative" to determine if the website is up. In the code below I've provided two different very simplistic implementations of this functionality.

The first is with a callback mechanism, and the second is with an async-await mechanism.

Here's a visual diagram of their entry and exit points for our purposes. Note that the callback arrow is pointed differently to show make it more obvious that it's a different type of exit point:

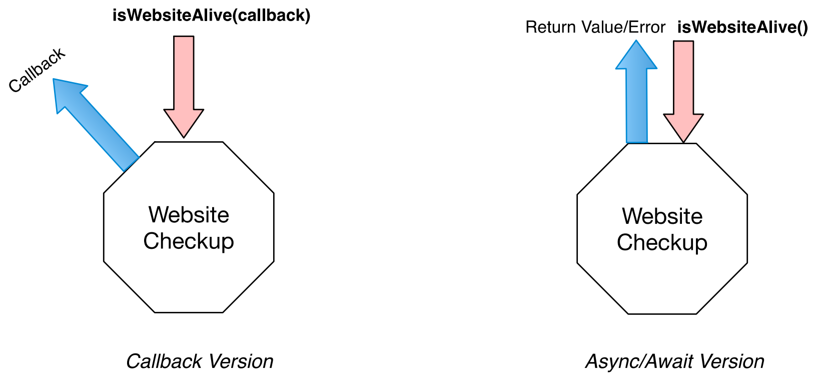


Figure : 6.1: IsWebsiteAlive callback vs async/await version

Here's the initial code. I'm assuming you know how promises work in JavaScript for this example.

If you need more info, I recommend reading https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

We're using `node-fetch` to get the URL's content:

Listing 6.1: IsWebsiteAlive vallback and await versions

```
//ch6-async/0-before/fetching-samples-before.js

const fetch = require("node-fetch");
//------CALLBACK VERSION---------------
const isWebsiteAliveWithCallback = (callback) => {
  const website = "http://example.com";
  fetch(website)
    .then((response) => {
      if (!response.ok) {
        //how can we simulate this network issue?
        throw Error(response.statusText); //A
      }
      return response;
    })
    .then((response) => response.text())
    .then((text) => {
      if (text.includes("illustrative")) {
        callback({ success: true, status: "ok" });
      } else {
        //how can we test this path?
        callback({ success: false, status: "text missing" });
      }
    })
    .catch((err) => {
      //how can we test this exit point?
      callback({ success: false, status: err });
    });
};
```

```
//------------------ AWAIT VERSION ---------------------
27    const isWebsiteAliveWithAsyncAwait = async () => {
28      try {
29        const resp = await fetch("http://example.com");
30        if (!resp.ok) {
31          //how can we simulate a non ok response?
32          throw resp.statusText; //A
33        }
34        const text = await resp.text();
35        const included = text.includes("illustrative");
36        if (included) {
37          return { success: true, status: "ok" };
38        }
39        // how can we simulate different website content?
40        throw "text missing";
41      } catch (err) {
42        throw { success: false, status: err }; //B
43      } };
46    module.exports = {
47      isWebsiteAliveWithCallback,
48      isWebsiteAliveWithAsyncAwait,
49    };
```

Things to Notice:

- A: In lines `09` and `32` we throw custom errors to have a single place where we handle problems in our code.
- B: In line `42` we throw an error to denote a failure to the user of our async/await function. This way we enable the user of the async/await version to use try/catch or then/catch syntax.

6.1.1 An initial attempt with an integration test

Since everything is hard coded into this functionality, if I were to ask people "how would you test this?" the initial reaction would usually revolve around writing an integration test. Here's how we could write an Integration test for the callback version:

Listing 6.2: An initial integration test

```
ch6-async/0-before/fetching-samples-before.integration.spec.js

const samples = require("./fetching-samples-before");

test("NETWORK REQUIRED (callback): website with correct content, returns true", (done) => {
  samples.isWebsiteAliveWithCallback((result) => {
    expect(result.success).toBe(true);
    expect(result.status).toBe("ok");
    done();
  });
});
```

To test a function whose exit point is a callback function, we pass it our own callback function in which we can:

- Check the correctness of the passed in values

- Tell the test runner to stop waiting through whatever mechanism is given to us by the test framework (in this case that's the `done()` function)

6.1.2 Waiting for the act

Because we're using callbacks as exit points, our test has to explicitly wait until the parallel execution completes. That parallel execution could be on the JavaScript event loop or it could be in a separate thread or even process if you're using another language.

In an arrange-act-assert pattern, our act part is the thing we need to wait out.

Most test frameworks will allow us to do so with special helper functions. In this case we can use the optional '`done`' callback that jest provides to signal that the test needs to wait until we explicitly call `done()`. if `done()` isn't called, our test will timeout and fail after the default 5 seconds (configurable of course).

jest has other means for async – We'll cover a couple of them later on in the chapter.

6.1.3 Integration testing async-await

What about the async-await version? We could technically write a test that looks almost exactly like the previous one since async-await is just syntactic sugar over promises:

Listing 6.3: Integration test with callbacks and .then()

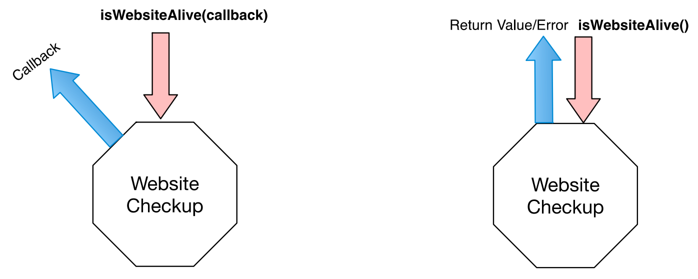
```
ch6-async/0-before/fetching-samples-before.integration.spec.js

test("NETWORK REQUIRED (.then): correct content, returns true",
                                               (done) => {
  samples.isWebsiteAliveWithAsyncAwait().then((result) => {
    expect(result.success).toBe(true);
    expect(result.status).toBe("ok");
    done();
  });
});
```

But there's no need to complicate our lives by forcing ourselves to use `done()` and `.then()` unnecessarily . We can use the `await` syntax in our test as well. This would force us to put the `async` keyword in front of the test function, but overall our test looks and reads much simpler:

Listing 6.4: Integration test with async/await

```
ch6-async/0-before/fetching-samples-before.integration.spec.js

test("NETWORK REQUIRED2 (await): correct content, returns true",
                                              async () => {
  const result = await samples.isWebsiteAliveWithAsyncAwait();
  expect(result.success).toBe(true);
  expect(result.status).toBe("ok");
});
```

In that regard, having async code that allows us to use async-await syntax turns our test into *almost* a regular run-of-the-mill value-based test. The entry point is also the exit point, as the first diagram showed earlier (repeated below).



Callback Version *Async/Await Version*

Even though the call is simplified, the call is still asynchronous underneath, which is why I still call this an integration test. What are the caveats for these types of tests? Let's discuss.

6.1.4 Challenges with integration tests

The tests we've just written aren't horrible as far as integration tests go. They're relatively short and readable, but still suffer from what any integration test suffers from:

- Lengthy run time. Compared to a unit test it is orders of magnitude slower, sometimes taking seconds or even minutes.)
- Flaky. The test can present inconsistent results (different timings based on where it runs, inconsistent failures or success etc.)
- Tests possibly irrelevant code and environment conditions. Tests multiple pieces of code that might be unrelated to what we care about (In our case it's the node-fetch library, network conditions, firewall, external website working etc)
- Longer investigations. When it fails, it requires a longer investigation/debugging due to many possible failure reasons.
- Simulation is harder. It is harder than it needs to be to simulate the negative test (simulating wrong website content, website down, network down etc..)
- Harder to trust results. We might believe the failure is due to an external issue when in fact it's a bug in our code. I'll talk about trust more in the next chapter.

Does all this mean you shouldn't write integration tests? Let me make it clear. I believe you should absolutely have integration tests, but I find that you don't need to have *many* of them to get enough confidence in your code. Whatever integration tests don't cover should be covered by lower level tests such as unit, api or component tests. I discuss this strategy in length in chapter 10 about testing strategies.

6.2 Making our code unit test friendly

How can we test the code we have as a unit test? I'll show some patterns that I like to do to make the code more unit-testable (i.e more easily inject or avoid dependencies, and check exit points).

- **Extract Entry Point.** Extracting the parts that are pure logic into their own function and treating those functions as entry points for our tests.
- **Extract Adapter:** Extracting the thing that is inherently asynchronous and abstracting it

away so that we can replace it with something that is synchronous.

6.2.1 Extracting a Logical Unit of Work

In this pattern, we take a specific piece of async work, and split it into the two pieces:

- The async part (which stays intact)
- The callbacks that are invoked when the async execution finishes. Those are extracted as new functions which eventually become entry points for a purely logical unit of work that we can invoke with pure unit tests.

The following diagram depicts this idea:

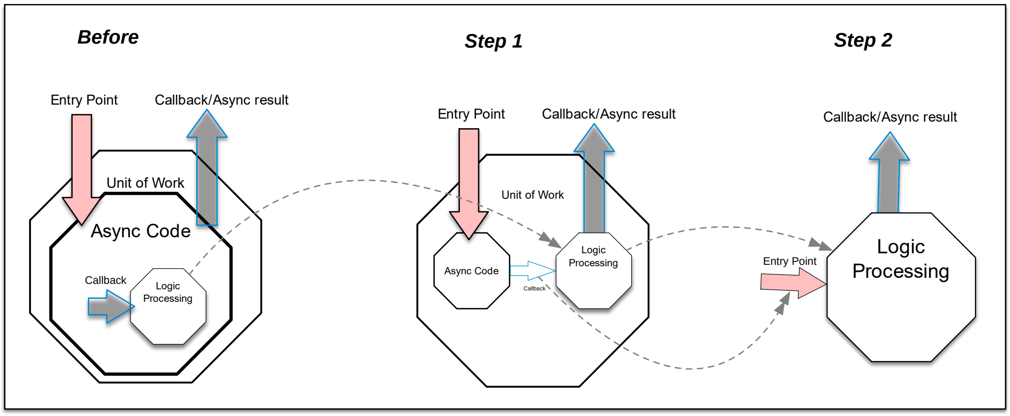


Figure 6.2: Extracting Entry Point Pattern

In the “**before**” diagram, we can see that we have a single unit of work that contains asynchronous code mixed in with logic that processes the async results internally and returns a result via a callback or promise mechanism.

In **step 1** we extract the logic into its own function(s) that contain only the results of the async work as inputs.

In **step 2** we externalize those functions so that we can use them as entry points for our unit tests.

This provides us with the important ability to test the logical processing of the async callbacks (and simulate inputs easily), while at the same time we can choose to write a higher level integration test against the original unit of work so we can gain confidence that the async “glue” works correctly as well.

If we had done only integration tests for all our scenarios, we would end up in a world of many long running and flaky tests. In the new world we’re able to have most of our tests as fast consistent tests, with a small layer of integration tests on top to make sure all the “glue” works in between, so we don’t sacrifice speed and maintainability over confidence.

6.2.2 Applying “Extract Entry Point” with Callbacks

Let’s apply this pattern to the code shown at the beginning of this chapter.

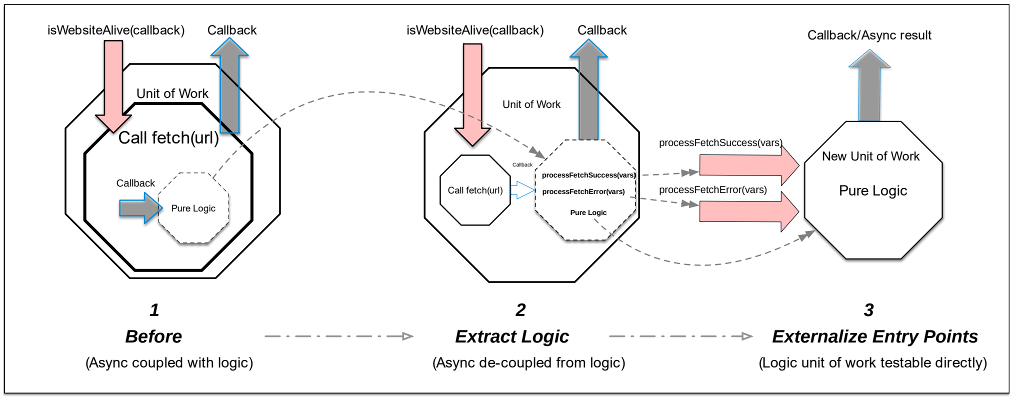


Figure 6.3: Extracting Entry Points from isWebsiteAlive()

1. We’ll extract any logical code that happens at the “edge” of the fetch results and put it in two separate functions: one for handling the success case, and the other for the error case.
2. We’ll then externalize these two functions so that we can invoke them direction from unit tests.

Here’s what the refactored code looks like:

Listing 6.5: extracting entry points with callback

```
ch6-async/1-fetch-callback/fetching-samples-callback.js
const fetch = require("node-fetch");

const isWebsiteAlive = (callback) => {
  fetch("http://example.com")
    .then(throwOnInvalidResponse)
    .then((resp) => resp.text())
    .then((text) => {
      processFetchSuccess(text, callback);
    })
    .catch((err) => {
      processFetchError(callback, err);
    });
};

const throwOnInvalidResponse = (resp) => {
  if (!resp.ok) {
    throw Error(resp.statusText);
  }
  return resp;
};

//Entry Point
```

```
23    const processFetchSuccess = (text, callback) => { //A
24      if (text.includes("illustrative")) {
25        callback({ success: true, status: "ok" });
26      } else {
27        callback({ success: false, status: "missing text" });
28      }
29    };
30
31    //Entry Point
32    const processFetchError = (err, callback) => { //A
33      callback({ success: false, status: err });
34    };
35
36    module.exports = {
37      isWebsiteAlive,
38      processFetchSuccess,
39      processFetchError,
40    };
41
```

A: New Entry Points.

As you can see in lines 23 and 32, at the higher level, the original unit of we started with now has three entry points instead of the one we had to start with. The new entry points can be used for unit testing, while the original one can still be used for integration testing:

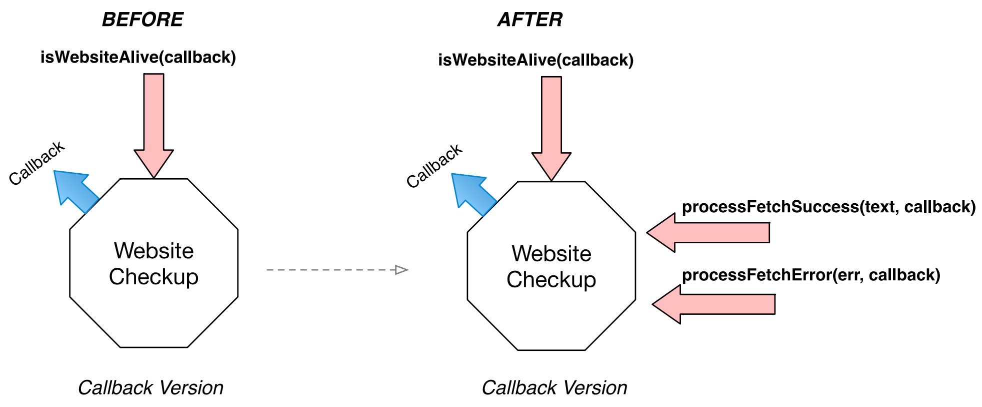


Figure 6.4: new entry points introduced

We’d still want an integration test for the original entry point written, but not more than one or two of those. Any other scenario can be simulated using the purely logical entry points, quickly and painlessly. Now we’re free to write unit tests that invoke the new entry points like this:

Listing 6.7: Unit tests with extracted entry points

```
ch6-async/1-fetch-callback/fetching-samples-callback.unit.spec.js
01    const samples = require("./fetching-samples-callback");
03    describe("Website alive checking", () => {
    test("content matches, returns true", (done) => {
      samples.processFetchSuccess("illustrative", (err, result) => {
        expect(err).toBeNull();
        expect(result.success).toBe(true);
        expect(result.status).toBe("ok");
        done();
      });
    });
    test("website content does not match, returns false", (done) => {
      samples.processFetchSuccess("bad content", (err, result) => {
        expect(err.message).toBe("missing text");
        done();
      });
    });
    test("When fetch fails, returns false", (done) => {
     samples.processFetchError(new Error("error text"),(err,result)=> {
        expect(err.message).toBe("error text");
        done();
      });
    });
  });
```

Several things to notice here:

- We're invoking the new entry points directly
- We're able to simulate various conditions easily
- Nothing is asynchronous in these test, but we still have a need for the `done()` function since the callbacks might not be invoked at all, and we'd want to catch that.
- We still need at least one integration test that gives us confidence that the asynchronous "glue" works between our entry points. That's where the original integration test can help, but we don't need to write all our test scenarios as integration tests anymore (again, most on this in chapter 10).

6.2.3 Extract Entry Point with Await

The same pattern we just applied can work well for standard async-await type function structures. Here's the visual representation of that refactoring:

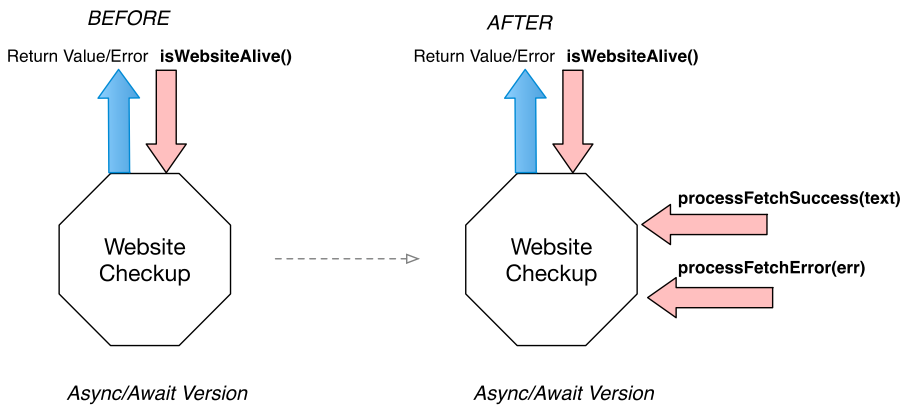


Figure 6.5: extract entry points with async/await

With providing an `async/await` syntax, we can go back to writing in a linear fashion with each line block the next line, that only returns values and throwing errors when needed.

Here's how that looks in our production code:

Listing 6.8: testing entry points extracted from async/await

```
ch6-async/2-fetch-await/fetching-samples-await.unit.spec.js

const samples = require("./fetching-samples-await");

describe("website up check", () => {
  test("on fetch success with good content, returns true", () => {
    const result = samples.processFetchContent("illustrative");
    expect(result.success).toBe(true);
    expect(result.status).toBe("ok");
  });
  test("on fetch success with bad content, returns false", () => {
    const result = samples.processFetchContent("text not on site");
    expect(result.success).toBe(false);
    expect(result.status).toBe("missing text");
  });
  test("on fetch fail, throws ", () => {
    expect(() => samples.processFetchError("error text"))
     .toThrowError("error text");
  });
});

```

Things to notice:

- Differently than the callback examples, we're using return or throw to denote success/failure. This is a common pattern of writing code in an async/await signature.

And our tests are simplified as well:

Listing 6.9: Unit tests with entry points and async/await

```
ch6-async/2-fetch-await/fetching-samples-await.unit.spec.js

const samples = require("./fetching-samples-await");

describe("website up check", () => {
  test("on fetch success with good content, returns true", () => {
    const result = samples.processFetchContent("illustrative");
    expect(result.success).toBe(true);
    expect(result.status).toBe("ok");
  });
  test("on fetch success with bad content, returns false", () => {
    const result = samples.processFetchContent("text not on site");
    expect(result.success).toBe(false);
    expect(result.status).toBe("missing text");
  });
  test("on fetch fail, returns error text and false", () => {
    const result = samples.processFetchError("error text");
    expect(result.success).toBe(false);
    expect(result.status).toBe("error text");
  });
});
```

Again, notice how we don't need to add any kind of `async/await` related keywords or be explicit about waiting for execution, because we've separated the logical unit of work from the asynchronous pieces that make our lives more complicated.

6.3 Extract Adapter Pattern

The extract adapter pattern takes the opposite view of the previous pattern.

We look at the asynchronous piece of code just like we look at any dependency discussed in the previous chapters -as something we'd like to replace in our tests to gain more control.

Instead of extracting the logical code into its own set of entry points, we're extracting the asynchronous code (our *dependency*) and abstracting it away under an adapter, which we can later inject just like any other dependency. The next figure shows this:

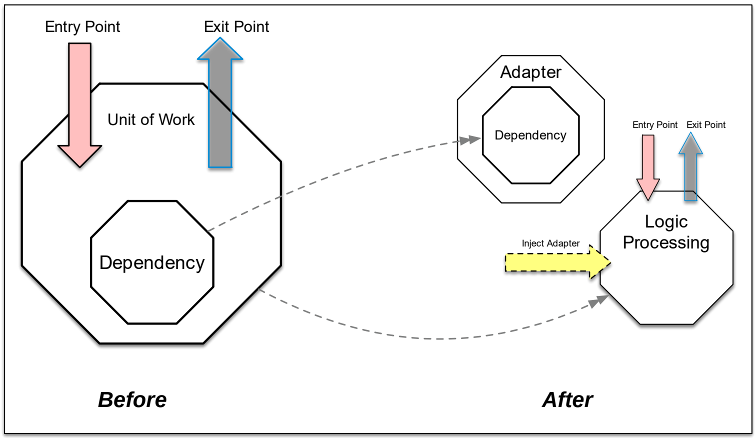


Figure 6.6: Extract Adapter Pattern

It is also common to create a special interface for the adapter that is simplified for the needs of the consumer of the dependency. Another name for this type of usage is the 'Interface Seggregation Principle" coined by Robert Martin. For example, a database dependency with dozens of functions might be hidden behind an adapter whose interface might only contain a couple of functions with custom names and parameters as to hide the complexity and simply both the consumer's code and the tests that simulate it. For more information Interface Segregation, see the following Wikipedia page: `https://en.wikipedia.org/wiki/Interface_segregation_principle`

In this case, we'll create a `network-adapter` that hides the real fetching functionality and has its own custom functions:

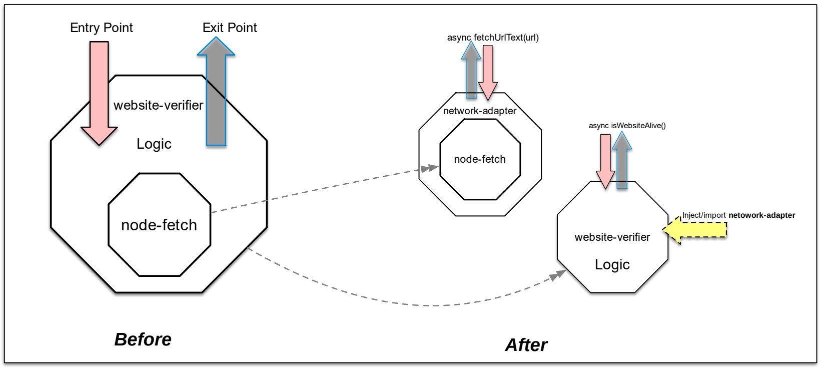


Figure 6.7: Extract network-adapter

Here's what the network-adapter looks like:

Listing 6.10: network-adapter's code

ch6-async/3-fetch-adapter-modular/network-adapter.js

```
const fetch = require("node-fetch");

const fetchUrlText = async (url) => {
  const resp = await fetch(url);
  if (resp.ok) {
    const text = await resp.text();
    return { ok: true, text: text };
  }
  return { ok: false, text: resp.statusText };
};

module.exports = {
  fetchUrlText,
};

```

Things to note:

- The `network-adapter` is the only file/module in the project that has an import/require to node-fetch. If that dependency changes at some point in the future, this increases that chances that the current file will be the only file that would need to change.
- We've simplified the function both by name and by functionality. We're hiding the need to fetch for the status and the text() from the url, and abstracting them both under a single easier to use function. This is an example of the interface segregation principle from uncle bob's SOLID principles.

Now we get to choose how to use the adapter. First we can see how we can use it in the modular style. Then we'll do functional and with a strongly typed interface (more object oriented).

6.3.1 Modular Adapter

Here's a modular usage of `network-adapter` by our initial `isWebsiteALive`:

Listing 6.11: isWebsiteLive using the network-adapter module

ch6-async/3-fetch-adapter-modular/website-verifier.js

```
const network = require("./network-adapter");

const isWebsiteAlive = async () => {
  try {
    const result = await network.fetchUrlText("http://example.com");
    if (!result.ok) {
      throw result.text;
    }
    const text = result.text;
    return processFetchSuccess(text);
  } catch (err) {
    throw processFetchFail(err);
  }
};

```

```
const processFetchSuccess = (text) => {
  ...
};

const processFetchFail = (err) => {
  ...
};

module.exports = {
  isWebsiteAlive,
};
```

Things to note:

- We're directly importing the `network-adapter` in our website-verifier module.
- We're only exporting the `isWebsiteAlive` function because we'll fake the `network-adapter` module in our tests later on.

Let's look at the unit tests for this module. Because we're using a modular design we can fake the module using `jest.mock()` in our tests. We'll also see how to inject the module in later examples, don't worry.

Listing 6.12: faking network-adapter with jest.mock

```
ch6-async/3-fetch-adapter-modular/website-verifier.unit.spec.js

jest.mock("./network-adapter"); // must be first line
const stubSyncNetwork = require("./network-adapter");
const webverifier = require("./website-verifier");

describe("unit test website verifier", () => {
  beforeEach(jest.resetAllMocks);

  test("with good content, returns true", async () => {
    stubSyncNetwork.fetchUrlText.mockReturnValue({
      ok: true,
      text: "illustrative",
    });
    const result = await webverifier.isWebsiteAlive(); //A
    expect(result.success).toBe(true);
    expect(result.status).toBe("ok");
  });

  test("with bad content, returns false", async () => {
    stubSyncNetwork.fetchUrlText.mockReturnValue({
      ok: true,
      text: "<span>hello world</span>",
    });
    const result = await webverifier.isWebsiteAlive(); //A
    expect(result.success).toBe(false);
    expect(result.status).toBe("missing text");
  });
...
```

A: Using await in our tests

Things to notice:

- Back to `await`
  - In lines `13` and `23` we're back to using `await` in our tests, because we are back to using the original entry point we started with at the beginning of the chapter.
  - Just because we're using `await`, doesn't mean our test is running asynchronously. Our test code, and the production code it invokes, actually run in a linear fashion, with an async-friendly signature.
  - We're naming our fake network as `stubSyncNetwork` to make the synchronous nature of the test clearer. Otherwise, it's harder to tell just by looking at the test if the code it invokes runs linearly or synchronously
  - We'll need to use `await` for the functional and object oriented designs as well, because the entry point requires it.
- Lots of Housekeeping needed to be able to fake `network-adapter`.
  - In line `01` : fake the `network-adapter` module
  - In line `02` : `require/import` it so that I can simulate its behavior
  - In line `06` : use `jest.resetAllMocks ()` to reset all the stubs before each test (or I can get weird behaviors in other tests)
  - This is due to the fact that I need to somehow override the direct import/require dependency my unit of work has on the network adapter.
- We're using jest's `mockReturnValue()` to simulate a return value from the stub module.

6.3.2 Functional Adapter

In the functional design pattern, the design of the `network-adapter` stays the same, however we enable its injection into our `website-verifier` differently – we add a new parameter to our entry point:

Listing 6.13: a functional injection design for website-verifier

```
//ch6-async/4-fetch-adapter-functional/website-verifier.js
01    const isWebsiteAlive = async (network) => {
02      const result = await network.fetchUrlText("http://example.com");
03      if (result.ok) {
04        const text = result.text;
05        return onFetchSuccess(text);
06      }
07      return Promise.reject(onFetchError(result.text));
08    };
09
       ...
22    module.exports = {
23      isWebsiteAlive,
24    };
```

Things to notice:

- We're expecting the `network-adapter` module to be injected through a common parameter to our function.

- In a functional design we can use high order functions and currying to configure a pre-injected function with our own network dependency. For our tests (in our tests we can simply send in a fake network via this parameter)
- As far as design of the injection goes, almost nothing else has changed from previous samples, other than the fact that we don't import the `network-adapter` module anymore. Reducing the amount of imports/requires can help maintainability in the long run.

Our tests are simpler (less boilerplate):

Listing 6.14: Unit test with functional injection of network-adapter

```
ch6-async/4-fetch-adapter-functional/website-verifier.unit.spec.js

01    const webverifier = require("./website-verifier");
02
03    const makeStubNetworkWithResult = (fakeResult) => {
04      return {
05        fetchUrlText: () => {
06          return fakeResult;
07        },
08      };
09    };
10    describe("unit test website verifier", () => {
11      test("with good content, returns true", async () => {
12        const stubSyncNetwork = makeStubNetworkWithResult({
13          ok: true,
14          text: "illustrative",
15        });
16        const result = await webverifier.isWebsiteAlive(stubSyncNetwork);
17        expect(result.success).toBe(true);
18        expect(result.status).toBe("ok");
19      });
20
21      test("with bad content, returns false", async () => {
22        const stubSyncNetwork = makeStubNetworkWithResult({
23          ok: true,
24          text: "unexpected content",
25        });
26        const result = await webverifier.isWebsiteAlive(stubSyncNetwork);
27        expect(result.success).toBe(false);
28        expect(result.status).toBe("missing text");
29      });
30      …
```

Things to notice:

- We don't need a lot of the boilerplate we had at the top of the file as we did in the modular design. We don't need to fake the module indirectly (`jest.mock(…)`), we don't need to re-import it for our tests (`require network-ad`apter), and we don't need to reset jest's state using `jest.resetAllMocks`)
- In line `02` , we have a new helper function `makeStubNetworkWithResult`. This helper returns a custom object that matches the important parts of interface of the `network-adapter`, which always returns our fake result. We call this function from each test to generate a new fake network adapter.

- In lines 16 and 26 we simply inject the fake network by sending it as a parameter to our entry point.

6.3.3 Object Oriented Interface Based Adapter

We've taken a look at the modular and functional designs. Let's turn our attention to the object oriented side of the equation.

In the object oriented paradigm, we can take the parameter injection we've done before and promote it into a constructor injection pattern.

Given the following network adapter and its interfaces (public API and results signature):

Listing 6.15: NetworkAdapter and its interfaces

```
ch6-async/5-fetch-adapter-interface-oo/INetworkAdapter.ts

export interface INetworkAdapter {
  fetchUrlText(url: string): Promise<NetworkAdapterFetchResults>;
}
export interface NetworkAdapterFetchResults {
  ok: boolean;
  text: string;
}


///ch6-async/6-fetch-adapter-interface-oo/network-adapter.ts

import fetch from "node-fetch";
import { INetworkAdapter, NetworkAdapterFetchResults }
        from "./INetworkAdapter";

export class NetworkAdapter implements INetworkAdapter {
  async fetchUrlText(url: string):
               Promise<NetworkAdapterFetchResults> {
    const resp = await fetch(url);
    if (resp.ok) {
      const text = await resp.text();
      return Promise.resolve({ ok: true, text: text });
    }
    return Promise.reject({ ok: false, text: resp.statusText });
  }
}

```

We can create a class `WebsiteVerifier` that has a constructor that receives an `INetworkAdapter` parameter:

Listing 6.16: WebsiteVerifier class with Constructor Injection

```
ch6-async/6-fetch-adapter-interface-oo/website-verifier.ts

import { INetworkAdapter, NetworkAdapterFetchResults }
from "./INetworkAdapter";
export interface WebsiteAliveResult {
  success: boolean;
  status: string;
}

export class WebsiteVerifier {
  constructor(private network: INetworkAdapter) {}

  isWebsiteAlive = async (): Promise<WebsiteAliveResult> => {
   let netResult: NetworkAdapterFetchResults;
   try {
   netResult = await this.network.fetchUrlText("http://example.com");
      if (!netResult.ok) {
        throw netResult.text;
      }
      const text = netResult.text;
      return this.processNetSuccess(text);
    } catch (err) {
      throw this.processNetFail(err);
    }
  };

  processNetSuccess = (text): WebsiteAliveResult => {
    const included = text.includes("illustrative");
    if (included) {
      return { success: true, status: "ok" };
    }
    return { success: false, status: "missing text" };
  };

  processNetFail = (err): WebsiteAliveResult => {
    return { success: false, status: err };
  };
}

```

The unit tests for this class can instantiate a fake network adapter and inject it through a constructor. In this case we're using Substitute.js to create a fake object thatfits the new interface:

Listing 6.17: Unit Tests for Object Oriented WebsiteVerifier

```
ch6-async/6-fetch-adapter-interface-oo/website-verifier.unit.spec.ts

import { WebsiteVerifier } from "./website-verifier";
import { Arg, Substitute } from "@fluffy-spoon/substitute";
import { INetworkAdapter, NetworkAdapterFetchResults }
from "./INetworkAdapter";

const makeStubNetworkWithResult = ( //A
  fakeResult: NetworkAdapterFetchResults
): INetworkAdapter => {
  const stubNetwork = Substitute.for<INetworkAdapter>(); //B
stubNetwork.fetchUrlText(Arg.any())
            .returns(Promise.resolve(fakeResult)); //C
  return stubNetwork;
};

describe("unit test website verifier", () => {
  test("with good content, returns true", async () => {
    const stubSyncNetwork = makeStubNetworkWithResult({
      ok: true,
      text: "illustrative",
    });
    const webVerifier = new WebsiteVerifier(stubSyncNetwork);

    const result = await webVerifier.isWebsiteAlive();
    expect(result.success).toBe(true);
    expect(result.status).toBe("ok");
  });

  test("with bad content, returns false", async () => {
    const stubSyncNetwork = makeStubNetworkWithResult({
      ok: true,
      text: "unexpected content",
    });
    const webVerifier = new WebsiteVerifier(stubSyncNetwork);

    const result = await webVerifier.isWebsiteAlive();
    expect(result.success).toBe(false);
    expect(result.status).toBe("missing text");
  });

  ...
```

A: Helper function our tests will use to simulate the network adapter.
B: We use substitute.js to generate the fake object
C: Make the fake adapter return what the test requires for a specific scenario.

This type of Inversion of Control and dependency injection work great together. In the object oriented world constructor injection with interfaces if very common and can, in many instances, provide a valid and maintainable solution for separating your dependencies form your logic.

6.4 Dealing with Timers

Timers such as `setTimeout` represent a very JavaScript specific problem. They are part of the domain and are used, for better or worse, in many pieces of code. On top of extracting adapters and entry points, sometimes it's just useful to disable and workaround them directly. We'll look at two patterns of getting around them directly:

- Directly Monkey patching the function
- Using Jest and other frameworks to disable and control them.

6.4.1 Stubbing timers out with Monkey Patching

Monkey patching is a way for a program to extend or modify supporting system software locally (affecting only the running instance of the program). Programing languages and runtimes such as JavaScript, Ruby and python allow monkey patching pretty easily. It's much more difficult to do with more strongly typed and compile time languages such as C# and Java.

Here's one way to do it in JavaScript. Given the following piece of code that uses a `setTimeout`:

Listing 6.18: Code we'd like to monkey patch with setTimeout

```
ch6-async/6-timing/timing-samples.js

const calculate1 = (x, y, resultCallback) => {
  setTimeout(() => { resultCallback(x + y); },
    5000);
};
```

We can monkey patch the `setTimeout` function to become synchronously literally setting that function's prototype in memory:

Listing 6.19: a simple monkey patching pattern

```
ch6-async/6-timing/timing-samples.spec.js

01    const Samples = require("./timing-samples");
02
03    describe("monkey patching ", () => {
04      let originalTimeOut;
05      beforeEach(() => (originalTimeOut = setTimeout));
06      afterEach(() => (setTimeout = originalTimeOut));
07
08      test("calculate1", () => {
09        setTimeout = (callback, ms) => callback();
10        Samples.calculate1(1, 2, (result) => {
11              expect(result).toBe(3);
12        });
13      });
14    });
```

Things to notice:

- Since everything is synchronous, we don't need to use `done()` to wait for a callback invocation.
- Lines 5 and 6 are responsible for saving and resetting `setTimout` before and after each test otherwise we could impact future tests implicitly.
- Line 9 replaces `setTimeout` with a purely synchronous implementation that invokes the received callback immediately.
- This requires a bunch of boilerplate code and is generally more error prone since we need to make sure to remember to cleanup correctly.

Now that we know this can work manually, let's look at what frameworks like jest provide us with top handle these situations:

6.4.2 Faking setTimeout with jest

Jest provides us with three major functions for handling most types of timers in our JavaScript code:

- `jest.useFakeTimers` : stubs out all the various timer functions such as `setTimetout`.
- `jest.ResetAllTimers` : Resets all fake timers to the real ones
- `jest.advanceTimerstoNextTimer` : Triggers any fake timer so that any callbacks are triggered.

Together, these functions help take care of most boilerplate code for us.
Here's the same test we just did, this time using jest's helper functions:

Listing 6.20: faking setTimeout with jest

```
ch6-async/6-timing/timing-samples.spec.js

describe("calculate1 - with jest", () => {
  beforeEach(jest.clearAllTimers);
  beforeEach(jest.useFakeTimers);

  test("fake timeout with callback", () => {
    Samples.calculate1(1, 2, (result) => {
      expect(result).toBe(3);
    });
    jest.advanceTimersToNextTimer();
  });
});
```

Things to notice:

- again, no need for done() since everything is synchronous.
- Without `advanceTimerstoNextTimer` our fake `setTimeout` will be stuck forever.
- `advanceTimerstoNextTimer` is also useful for scenarios such as one where the module being tested schedules a `setTimeout()` whose callback schedules another `setTimeout()` recursively (meaning the scheduling never stops). In these scenarios, it's useful to be able to run forward in time by a single step at a time.
- With `advanceTimerstoNextTimer` you could potentially advance all timers by a specified amount of steps to simulate the passage of “steps” that will trigger the next timer callback waiting in line.

The same pattern also works great with `setInterval`:

Listing 6.21: a function that uses setInterval

```
ch6-async/6-timing/timing-samples.js

const calculate4 = (getInputsFn, resultFn) => {
  setInterval(() => {
    const { x, y } = getInputsFn();
    resultFn(x + y);
  }, 1000);
};
```

In this case our function takes in two callbacks as parameters. One that provides the inputs to calculate, and the other to callback with the calculation result. It uses `setInterval` to continuously get more inputs and to calculate their results.

Here’s a test that will advance fake and trigger the interval twice and expect the same result from both invocations.

Listing 6.22: advancing fake timers in a unit test

```
ch6-async/6-timing/timing-samples.spec.js

describe("calculate with intervals", () => {
  beforeEach(jest.clearAllTimers);
  beforeEach(jest.useFakeTimers);
04
05    test("calculate, incr input/output, calculates correctly", () => {
06      let xInput = 1;
07      let yInput = 2;
08      const inputFn = () => ({ x: xInput++, y: yInput++ });
09
10      const results = [];
11      Samples.calculate4(inputFn, (result) => results.push(result));
12
13      jest.advanceTimersToNextTimer(); //A
14      jest.advanceTimersToNextTimer();
15
16      expect(results[0]).toBe(3);
17      expect(results[1]).toBe(5);
18    });
19  });
```

Things to Notice:

- We call `advanceTimersToNextTimer` twice to invoke `setInterval` twice
- We do this to verify we are calculating the new values being passed in correctly
- We could have written the same test with only a single invocation and a single expect and would have gotten close to the same amount of confidence this more elaborate test provides.
- In line 8 we're adding to a variable so that we can verify the number of callbacks. By doing that we've sacrificed some readability and complexity to make the test a bit more robust. I'm not crazy about it, but I've done this from time to time when I needed more confidence.

6.5 Dealing with Common Events

I can't talk about async unit testing and not discuss basic events flow. In many ways this code hopefully now seems straightforward, but I want to make sure we go over it explicitly. Just in case.

6.5.1 Dealing with event emitters

To make sure we're all on the same page, here's a clear and concise definition of event emitters as found at https://www.digitalocean.com/community/tutorials/using-event-emitters-in-node-js :

EVENT EMITTERS

"In Node.js, Event emitters are objects that trigger an event by sending a message that signals that an action was completed.

JavaScript developers can write code that listens to events from an event emitter, allowing them to execute functions every time those events are triggered.

These events are composed of an identifying string and any data that needs to be passed to the listeners."

Consider the following Adder class, that emits an event every time it adds something:

Listing 6.23: A simple event emitter based Adder

```
ch6-async/7-events/adder-emitter/adder.js
```

```
const EventEmitter = require("events");

class Adder extends EventEmitter {
  constructor() {
    super();
  }

  add(x, y) {
    const result = x + y;
    this.emit("added", result);
    return result;
  }
}

module.exports = {
  Adder,
};
```

The simplest way to write a unit test that verifies the event is emitted is to literllay subscribe to the event in our test, and verify it triggers when we call the add() function:

Listing 6.24: Testing an event emitter by subscribing to it

```
//ch6-async/7-events/adder-emitter/adder.spec.js
const { Adder } = require("./adder");
describe("events based module", () => {
  describe("add", () => {
    it("generates addition event when called", (done) => {
      const adder = new Adder();
      adder.on("added", (result) => {
        expect(result).toBe(3);
        done();
      });
      adder.add(1, 2);
    });
  });
});
```

Things to notice:

- By using done() we are verifying the event actually got emitted. If we didn't, in case the event wasn't emitted, our test would simply pass because the subscribed code never executed.
- By adding `expect(x).toBe(y)` we are also verifying the values sent in the event parameters, as well as implicitly testing that the event was triggered.

6.5.2 Dealing with Click events

What about those pesky UI events such as click()? How can we test that we have bound them correctly via our scripts? Here's a simple web page and its associated logic:

Listing 6.25: A simple web page with javascript click functionality

```
//ch6-async/7-events/click-listener/index.html
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <title>File to Be Tested</title>
    <script src="index-helper.js"></script>
</head>
<body>
    <div>
        <div>A simple button</div>
        <Button data-testid="myButton" id="myButton">
 Click Me
</Button>
        <div data-testid="myResult" id="myResult">Waiting...</div>
    </div>
</body>
</html>
```

Listing 6.26: The logic for the web page in JavaScript

```
//ch6-async/7-events/click-listener/index-helper.js

window.addEventListener("load", () => {
  document
    .getElementById("myButton")
    .addEventListener("click", onMyButtonClick);

  const resultDiv = document.getElementById("myResult");
  resultDiv.innerText = "Document Loaded";
});

function onMyButtonClick() {
  const resultDiv = document.getElementById("myResult");
  resultDiv.innerText = "Clicked!";
}

```

We have a very simplistic piece of logic that makes sure our button sets a special message when clicked. How can we test this?

Here's an antipattern: we could subscribe to the click event in our tests and make sure it was triggered – but this would provide no value to us. What we care about is that the click has actually done something useful other than triggering.

Here's a better way: we can trigger the click event and make sure it has changed the correct value inside the page – this would provide real value. Here's a diagram that shows this:

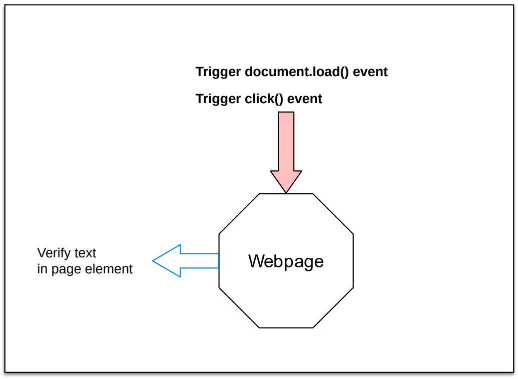


Figure 6.8: Click as an entry point, element as an exit point.

Here's what our test might look like:

Listing 6.27: triggering a click event and testing an element's text

ch6-async/7-events/click-listener/index-helper.spec.js

```
/**
 * @jest-environment jsdom
 */
//(the above is required for window events)
const fs = require("fs");
const path = require("path");
require("./index-helper.js");

const loadHtml = (fileRelativePath) => {
  const filePath = path.join(__dirname, "index.html");
  const innerHTML = fs.readFileSync(filePath);
  document.documentElement.innerHTML = innerHTML;
};

const loadHtmlAndGetUIElements = () => {
  loadHtml("index.html");
  const button = document.getElementById("myButton");
  const resultDiv = document.getElementById("myResult");
  return { window, button, resultDiv };
};

describe("index helper", () => {
  test("vanilla button click triggers change in result div", () => {
    const { window, button, resultDiv } = loadHtmlAndGetUIElements();
    window.dispatchEvent(new Event("load"));

    button.click();

    expect(resultDiv.innerText).toBe("Clicked!");
  });
});
```

Things to notice:

- Jsdom: In lines 1-3 we're applying the browser-simulating jsdom environment just for this file. It's the default for jest, but my other example tests are using the "node" environment so this was required. You can find out more about jsdom at https://github.com/jsdom/jsdom
- In lines 9 and 15 I've extracted two utility methofs: loadHtml and loadHtmlAndGetUIElements(), so that I can write cleaner, more readable tests, and be able to have less issues changing my tests if UI item locations or IDs change in the future.
- In line 25 we're simulating the `document.load` event so that our custom script under test can start running.
- In line 27 we're triggering the `click` , as if the used had clicked the button.
- In line 29 we're verifying that an element in our document had actually changed, which means our code successful subscribed to the event and had done its work.
- Notice how we don't actually care about the underlying logic inside the index helper file. We just rely on observed state changes in the UI which acts as our final exit point. This allows less coupling in our tests , so that if our code under test changes, we are less likely to change the test unless observable (publicly noticeable) functionality has truly changed.

6.6 Bringing in dom-testing-library

Our test has a lot of boilerplate vode, moslty for finding elements and verying their contest. I recommend looking into the open source dom testing library written by Kent C. Dodds. This library also has variants applicable to most JS frontend frameworks today such as React, Angular and Vue.js. We'll be using the "vanilla" version of it named "dom testing library".

What I like about it is that it aims to allow us to write tests closer to the point of view of the user interacting with our web page. Instead of using IDs for elements, we query by elemt text,, firing events is a bit cleaner, and querying and waiting for elements to appear or disapper I cleaner and hidden under syntactic sugar. It's quite useful once you use it in multiple tests.

Here's how our test looks with this library:

Listing 6.28: using dom testing library in a simple test

```
ch6-async/7-events/click-listener/index-helper-testlib.spec.js

01    /**
02     * @jest-environment jest-environment-jsdom-sixteen
03     */
04    //(the above is required for window events)
05    const fs = require("fs");
06    const path = require("path");
07    const { fireEvent, findByText, getByText }
         = require("@testing-library/dom");
08    require("./index-helper.js");
09
10    const loadHtml = (fileRelativePath) => {
11      const filePath = path.join(__dirname, "index.html");
12      const innerHTML = fs.readFileSync(filePath);
13      document.documentElement.innerHTML = innerHTML;
14      return document.documentElement;
15    };
```

```
16
17    const loadHtmlAndGetUIElements = () => {
18      const docElem = loadHtml("index.html");
19      const button = getByText(docElem, "click me", { exact: false });
20      return { window, docElem, button };
21    };
22
23    describe("index helper", () => {
24      test("dom test lib button click triggers change in page", () => {
25        const { window, docElem, button } = loadHtmlAndGetUIElements();
26        fireEvent.load(window);
27
28        fireEvent.click(button);
29
30        //wait until true or timeout in 1 sec
31        expect(findByText(docElem,
                          "clicked",
                          { exact: false })).toBeTruthy();
32      });
33    });
```

Things to notice:

- In line 07 we import some of the library APIs to be used
- In line 14 we're actually returning the document element, since the library APIs require using that as the basis for most of the work.
- In line 26, and 28 we use the library's `fireEvent` api to simplify event dispatching.
- In line 31 we're using the "`findByText`" query that will wait until item is found or time out within 1 second.
- In lines 31 and 19 We're using the regular text of the page items to get the items, instead of their IDs or test ids. This is part of the way the library pushes us to work so things feel more natuaral and from the user's point of view.
- To make the test more sustainable over time we're using the "exact:false" flag so that we don't have to worry about upper casing issues or missing letter at the start or end of the string. This removes the need to change the test on small text changes that are less important.

6.7 Summary

This has been a whirlwind tour of async unit testing in JavaScript. We've covered quite a bit of ground but the most important things to take away are:

- Consider when to extract new entry points makes sense for you
- Consider extracting an adapter to remove a dependency from a piece of code.
- Sometimes you can't get out of just faking out the various timers with monkey patching or a framework.
- Events testing is really easy once you get the hang of it.
- What about Observables? If you're interested in unit testing with RxJS and observables, I've put some ideas on testing observables in the appendix at the end of the book.

7 Trustworthy Tests

This chapter covers

- **How to know you trust a test**
- **Detecting untrustworthy failing tests**
- **Detecting untrustworthy passing tests**
- **Dealing with flaky tests**

No matter how you organize your tests, or how many you have, they're worth very little if you can't trust them, maintain them, or read them. The tests that you write should have three properties that together make them good:

- *Trustworthiness*—Developers will want to run trustworthy tests, and they'll accept the test results with confidence. Trustworthy tests don't have bugs, and they test the right things.
- *Maintainability*—Unmaintainable tests are nightmares because they can ruin project schedules, or they may be sidelined when the project is put on a more aggressive schedule. Developers will simply stop maintaining and fixing tests that take too long to change or that need to change very often on very minor production code changes.
- *Readability*—This means not just being able to read a test but also figuring out the problem if the test seems to be wrong. Without readability, the other two pillars fall pretty quickly. Maintaining tests becomes harder, and you can't trust them anymore because you don't understand them.

This chapter and the next two present a series of practices related to each of these pillars that you can use when doing test reviews. Together, the three pillars ensure your time is well used. Drop one of them, and you run the risk of wasting everyone's time.

Trust is the first of the three pillars that I like to categorize good unit tests on. It's fitting that we start with it. If we don't trust the tests, what's the point in running them? What's the point in fixing them or the code if they fail? What's the point of maintaining them?

7.1 How to know you trust a test

What does "trust" mean for a software developer in the context of a test? Perhaps it's easier to explain based on what we do or don't do when a test fails or passes. More specifically,

You might not trust the test If..

- It fails and you're not worried (You believe it's a "false negative")

  You (or someone else tells you it's OK to) feel like it's OK to ignore the results of this test (either because it passes every once in a while or because you feel it's not relevant or buggy.
- It passes and you are worried (You believe it's a "false positive")

You still feel the need to manually debug or test the software "just in case".

You might trust the test If..

- The test fails and you're genuinely **worried** that something broke. You don't move on assuming the test is wrong.
- The test passes and you feel relaxed, not feeling the need to test or debug manually.

In the next few sections I'll cover the following:

- Looking at test failures as a source to find untrustworthy tests
- Looking at passing tests' code to detect untrustworthy test code
- A few generic practices that can enhance trustworthiness in tests

7.2 Why Tests Fail (and how it can help us)

Ideally, your tests (any test, not just unit test) should only be failing for *a good reason*. That good reason is of course – a real bug was uncovered in the underlying production code.

Unfortunately, tests can fail on a multitude of reasons. We can collectively assume that any reason other than that one good reason should trigger our "untrustworthy!" warning, but it's not all tests fail the same way, and recognizing the reasons tests can fail can help us build a roadmap in our head of what we'd like to do in each case.

Here are some of the reasons that tests fail:

Except the first point here, all the rest are the test telling you it should not be trusted in its current form. Let's go through them.

7.3 A real bug has been uncovered in the production code. (good!)

- A test fails because there is a bug in production code.
- That's good! That's why we have tests.

Let's move to other reasons tests fail.

7.4 Buggy (lying) test

Test fails because the test is buggy.

The production code might (potentially) be correct but that doesn't matter, because the test itself has a bug which causes the test to fail.

It could be that we're asserting on the wrong expected result of an exit point, or that we're using the system under test wrong. It could be that we're setting up the context for the test wrong, or that we misunderstand what we were supposed to test.

Either way, a buggy a test can be quite dangerous, because a bug in a test can also cause it to *pass* and leave us unsuspecting of what's really going on. (we'll talk more about tests that don't fail but should make us suspicious later in the chapter) .

7.4.1 How to recognize a buggy (lying) test

You have a failing test, but you might have already debugged the production code and couldn't find any bug there. This is the point where you should start suspecting the failing test.

There's no way around it. You're going to have to slowly debug the test code.

You might find you're

- Asserting on the wrong thing or on the wrong exit point
- Injecting a wrong value into the entry point
- Invoking the entry point wrongly
- Some other small mistake that happens when you write code at 2AM. (that's not a sustainable coding strategy, by the way. Stop doing that.)

7.4.2 What do we do once we've established a buggy test?

1. Don't panic. This might be the millionth time you've found one of these so you might be panicking and thinking "our tests suck". You might also be right about that. But does that mean you should panic? No.
2. Fix the bug in the test.
3. Run the test and see if it now passes.
4. **If the test passes don't be happy too soon:**
   a) Go to the production code and place an obvious bug that should be caught by our newly fixed test. (for example, change a boolean to always be true. Or false. It's up to you).
   b) Run the test again and make sure it fails. If it doesn't, you might still have a bug in your test. Fix test until it can find the production bug and see it fail now.
   c) OK, test is failing for an obvious production code issue. Now *fix* the production code issue you just made and run the test again. It should pass.
   d) If the test is now passing, you're done. You've now seen the test passing and failing when it should. Commit the code and move on.
   e) If the test is still failing it might still have another bug in the test. Repeat from step (b)

If you went to step (4) and the test is failing, it could mean the test still has a bug. Fix the test until the bug is gone and it's passing.

If the test is still failing, you might have come across a real bug in production code. In which case, good for you!

7.4.3 Reducing or avoiding buggy tests in the future

One of the best ways I know in detecting and preventing buggy tests is to write your code in a test driven manner. I've explained a bit more about this technique in chapter 1 of this book. I also practice this technique in real life.

TDD (for short) allows us to see both states of a test: both that it fails when it should (that's the initial state we start in) and that it passes when it should (when the production code under test is written to make the test pass). If the test stays failing, we find a bug in the test. If the test starts our passing, we have a bug in the test.

Another great way to reduce the likelihood of a logical bug in a test is to remove the amount of logic related code that exists in the test.

7.4.4 Avoiding logic in unit tests

The chances of having bugs in your tests increase almost exponentially as you include more and more logic in them. I've seen plenty of tests that should have been simple become dynamically logic-changing, random-number-generating, thread-creating, file-writing monsters that are little test engines in their own right. Sadly, because they were "tests", the writer didn't consider that they might have bugs or didn't write them in a maintainable manner. Those test monsters waste more time to debug and verify than they save.

But all monsters start out small. Often, an experienced developer in the company will look at a test and start thinking, "What if we made the function loop and create random numbers as input? We'd surely find lots more bugs that way!" And you will, especially in your tests.

Test bugs are one of the most annoying things for developers, because you'll almost never search for the cause of a failing test in the test itself. I'm not saying that such tests don't have any value. In fact, in some special situations I'm likely to write such tests myself. I try to avoid this practice as much as possible.

If you have any of the following inside a unit test, your test contains logic that I usually recommend be reduced or removed completely:

- `switch`, `if`, or `else` statements
- `foreach`, `for`, or `while` loops
- Concatenations ('+' etc..)
- Try-catch

7.4.5 Logic in asserts – creating dynamic expected values

Here's a quick example of the concatenation to start us off:

Listing 7.1: a test with logic in it

```
ch7-trust/trust.spec.js

  describe("makeGreeting", () => {
    it("returns correct greeting for name", () => {
      const name = "abc";
      const result = trust.makeGreeting(name);
      expect(result).toBe("hello" + name);
    });
```

To understand the problem with this test, here is the code being tested. Notice that the "+" sign makes an appearance in both.

Listing 7.2: code under test

```
ch7-trust/trust.js

const makeGreeting = (name) => {
  return "hello" + name;
};
```

Notice how the algorithm (very simplistic but still an algorithm) of connecting a name with a "`hello`" string is repeated in both the test and the code under test:

```
expect(result).toBe("hello" + name);  //← out test
return "hello" + name;  // ← out code under test
       My issue with this test is that the algorithm under test is repeated in the test itself.
       Which means if the algorithm has a but in the code under test, the test also contains the
       same bug. Which means that it will not catch the bug, but instead expect the incorrect
       result from the code under test.
```

In this case the incorrect result is that we're missing a space character between the concatenated word – but hopefully you can see how the same issue can become much more complex with a more complex algorithm.

This is a trust issue because we can't trust this test to tell us the truth, since its logic is a repeat of the logic being tested. The test might pass when the bug exists in the code. We can't trust the test's result.

To generalize this rule:

Avoid dynamically creating the expected value in your asserts, use hardcoded values when possible.

A more trustworthy version of this test can be rewritten as in listing 7.3:

Listing 7.3: a more trustworthy test

```
ch7-trust/trust.spec.js

it("returns correct greeting for name 2", () => {
  const result = trust.makeGreeting("abc");
  expect(result).toBe("hello abc");
});
```

Because the inputs in this test are so simple, it's very easy to write a hardcoded expected value – and that's what I usually recommend – make the test inputs so simple that it is trivial to create an expected hardcoded version of the expected value. (This is mostly true for unit tests. For higher level tests this is a bit harder , so you might not be able to do this – which is another reason why higher level tests should be considered a bit more risky – they often have dynamically expected results - but try to avoid this anytime you have a chance to).

"But Roy" , you might say, *"Now we are repeating ourselves – the string "abc" is repeated twice. We were able to avoid this in the previous test"*. When push comes to shove, trust should trump maintainability. What good is a highly maintainable test that I cannot trust?

7.4.6 Other forms of logic

Here's the opposite case of logic by creating the inputs dynamically (using a loop), forces us to dynamically decide what the expected output should be.

Given the following code to test:

Listing 7.4: a name finding function

```
ch7-trust/trust.js

const isName = (input) => {
  return input.split(" ").length === 2;
};
```

Here's a clear anti pattern for a test:

Listing 7.5: loops and ifs in a test

```
ch7-trust/trust.spec.js

01    describe("isName", () => {
02      const namesToTest = ["firstOnly", "first second", ""];
03
04      it("correctly finds out if it is a name", () => {
05        namesToTest.forEach((name) => {
06          const result = trust.isName(name);
07          if (name.includes(" ")) {
08            expect(result).toBe(true);
09          } else {
10            expect(result).toBe(false);
11          }
12        });
13      });
14    });
```

Our main culprit here is in line 2 – we've decided to test multiple inputs using an array.

This forces us to:

- Loop over the array (loops can have bugs too)
- Because we have different scenarios for the values (with and without spaces) we need an if-else to know what the assertion is expecting (if-else can have bugs too)
- We are repeating a part of the production algorithm (expecting a space) which brings us back to the previous concatenation example and it's problems.
- Out test name is much more generic – since have are accounting for multiple scenarios and expected outcomes we can only say that it "works" – that's bad for readability.

All around – this is a majorly bad decision in my mind. It's better to separate this into 2 or 3 tests , each their their own scenario and name, but more importantly ,it would allow us to use hardcoded inputs and assertions , and remove any loops and if-else's from the code.

Anything more complex causes the following problems:

- The test is harder to read and understand.
- The test is hard to re-create. (Imagine a multithreaded test or a test with random numbers that suddenly fails.)
- The test is more likely to have a bug or to test the wrong thing.
- Naming the test may be harder because it does multiple things.

Generally, monster tests replace original simpler tests, and that makes it harder to find bugs in the production code. If you must create a monster test, it should be added as a new test and not replace existing tests, and it should reside in a project or folder explicitly titled to hold non unit tests (sometimes I call this "integration tests" sometimes "complex tests" and try to keep the number to a minimum acceptable amount).

Anyway, we were discussing why tests fail , and particularly, we were discussing buggy tests. But a test can also fail if it out of date.

7.4.7 Even more logic

Logic might be found not only in tests but also in **test helper methods, handwritten fakes, and test utility classes**. Remember, every piece of logic you add in these places makes the code that much harder to read and increases the chances of your having a bug in a utility method that your tests use.

If you find that you need to have complicated logic in your test suite for some reason (though that's generally something I do with integration tests, not unit tests), at least make sure you have a couple of tests against the logic of your utility methods in the test project. It will save you many tears down the**Error! Bookmark not defined.** road

7.5 Out of Date test (feature changed)

A test can fail if it's no longer compatible to the current feature that's being tested. Say you have a login feature. In earlier version to login you needed to provide username, and a password. In a newer version, a 2-factor authentication scheme replaced the old login. This means the existing test will start failing because it's not providing all the right parameters to the login functions.

7.5.1 What can we do now?

We now have two options:

- Adapt the test to the new functionality
- Write a new test for the new functionality and remove the old test because it has now become irrelevant.

7.5.2 Avoiding or preventing this in the future

Things change. I don't think it's possible to *not* have out-of-date tests at some point in time. We deal with change in the next chapter, relating to maintainability of tests, and how well they can handle changes in the application.

7.6 Conflicting test (with another test)

Given two tests, one of them fails and one of them passes, but they cannot pass together.

You'll usually only see the failing test, because the passing one is… well, passing.

The failing test fails because it suddenly conflicts with a new behavior of the application. On the other hand, a conflicting test exists that does expect the new functionality to work.

For example:

- Test A verifies that calling a function with two parameters produces "3".
- Test B verifies that calling a function with the same two parameters produces "4"

7.6.1 What can we do now?

The root cause is that one of the tests has become irrelevant, which means it needs to be removed. Which one should be removed? That is a question we'd need to ask a product owner, because the answer is related to which functionality is the correct functionality expected from the application.

7.6.2 Avoiding this in the future

I feel this is a healthy dynamic and I'm fine with not avoiding it.

7.7 Flaky Test

A test can fail inconsistently. Even if the production code under test hasn't changed, a test can suddenly fail without any provocation, then pass again, then fail again. We call a test like that "flaky".

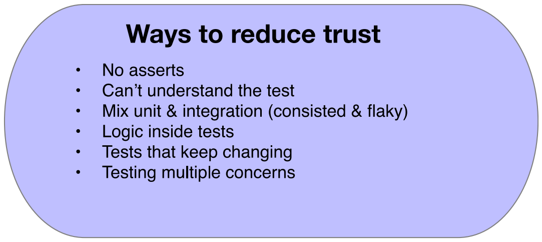
Flaky tests are a special beast and I deal with it at the last section of this chapter.

7.8 Smelling a false sense of trust in passing tests

OK. We've now covered failed tests as a source of detecting tests we shouldn't trust. What about all those quiet, green tests we have lying all over the place? Should we trust them? What about a test we need to do a code review for before it's pushed into a main branch? What should we look for?

Let's call the idea that you trust a test but you really shouldn't, but you don't know it yet – "false-trust" . In that you trust the test, but it is not worthy of your trust – you just don't know it yet.

Being able to review tests and find possible false-trust issues has immense value because not only are you able to fix those tests for yourself, you're affecting the trust of everyone else who's ever going to read or run those tests. Here are some reasons I reduce my trust in tests even if they are passing:



Ways to reduce trust

- No asserts
- Can't understand the test
- Mix unit & integration (consisted & flaky)
- Logic inside tests
- Tests that keep changing
- Testing multiple concerns

7.8.1 Tests that don't assert anything

We all agree that a test that doesn't actually verify that something is true or false is less than helpful, right? Less than helpful because it also costs in maintenance time, refactoring and readability time and sometimes unnecessary "noise" if it needs changing due to API changes in production code.

If you do see a test with no asserts consider:

- There may be hidden asserts in some function call (this causes a readability problem if the function is not named to explain this)
- Sometimes people write a test that exercises a piece of code simply to make sure that the code does not throw an exception. This does have some value – and if that's the test you choose to write, make sure that the test name somewhere contains this fact such as "does not throw". To be even more specific, many test apis support the ability to specificy that something does not throw an exception. This is how you can do this in jest:

```
expect(() => someFunction()).not.toThrow(error)
```

By the way – if you do have such tests, make sure they is a very small amount of them. I don't recommend this to be a standard, but only for really special cases.

- Sometimes people simply forget to write an assert due to lack of experience. Consider adding the missing assert or removing the test if it brings no value.
- Sometimes people actively write tests to achieve some imagined "test coverage" goal set by management. Those tests usually serve no real value except to get management off people's backs so they can do real work. Show your manager the following text:

  Managers: Code coverage shouldn't ever be a goal on its own. It doesn't mean "code quality". In fact, it often causes your developers to write meaningless tests that will cost even more time to maintain. Instead, measure "Escaped Bugs" ,"Time to Fix" and well as other metrics that I provide in chapter 11.

7.8.2 Not understanding the tests

This is a huge issue and I'll deal with it deeply in chapter 9 about readability. Some of these issues are:

- Tests can have bad names
- Tests can be too long or have convoluted code
- Variable names can be confusing
- Tests might contain hidden logic or assumptions that cannot be understood easily
- Test results are inconclusive (not failed or passed)
- Test messages do not provide enough information

We don't understand the test that's failing or passing – so we don't know if we should feel worried or not.

7.8.3 Mixing Unit & Integration (Flaky) Tests

They say that one rotten apple spoils the bunch. The same is true for flaky tests mixed in with non flaky tests. Integration tests are much more likely to be flaky than unit tests because they have more dependencies (see last section in this chapter for a deeper dive on test levels).

If you find that you have a mix of integration and unit tests in the same folder or test execution command, you should be suspicious.

Us humans like to take the path of least resistance, and when it comes to code that hasn't changed much. If a developer runs all the tests and one of them fails, if there was a way to blame a "missing configuration" or "a network issue" instead of investigating or spend time on fixing a real problem, and that's *especially* true if you're under serious time pressure, over committed to delivering things you're already late on.

The easiest thing is to blame any failing test as a flaky test. Because flaky and non flaky tests are mixed up with each other, that's a simple thing to do and a good way to ignore the issue and work on something more fun. Because of this human factor, we'd like to remove the option to "blame" a test as a flaky one. What should you do prevent this? Aim to have a "safe green zone". (see sidebar)

The safe green zone

Write (or move) your integration and unit tests in separate places. By doing that, you give the developers on your team a safe green test area that contains only non flaky, fast tests, where they know that they can get the latest code version, they can run all tests in that namespace or folder, and the tests should all be green (given no changes to production code). If some tests in the safe green zone don't pass, a developer is much more likely to feel concern.

An added benefit to this separation is that developers are more likely to run our unit tests more often, now that the run time is faster without the integration tests. Better to have some feedback than no feedback, right?

The automated build pipeline should take care of running any of the "missing" feedback tests that developers can't or won't run on their local machines, but at least they ran the fast ones.

7.8.4 Having logic inside unit tests

I've covered this anti pattern at the beginning of this chapter in section 7.4.

7.8.5 Testing multiple concerns (entry/exit points)

A **Error! Bookmark not defined.**concern is an *exit point*, as explained in chapter 1. It is a single end result from a unit of work: a return value, a change to system state, or a call to a third-party object.

Here's a simplistic example. Given the following function that has two exit points, or two concerns (it both returns a value and triggers a passed in callback function):

```
const trigger = (x, y, callback) => {
  callback("I'm triggered");
  return x + y;
};
```

We could write a test that checks both of these exit points in the same test:

```
describe("trigger", () => {
  it("works", () => {
    const callback = jest.fn();
    const result = trigger(1, 2, callback);
    expect(result).toBe(3);
    expect(callback).toHaveBeenCalledWith("I'm triggered");
  });
});
```

The first reason testing more than one can backfire is that your test name suffers. I discuss readability in chapter 9, but a quick note on naming – naming tests is hugely important for both debugging and documentation purposes. I spend a lot fo time thinking about a good name for a test, and I’m not ashamed to admit it.

Naming a test may seem like a simple task, but if you’re testing more than one thing, giving the test a good name that indicates what’s being tested becomes annoying and clumsy. Often you end up with a very generic test name that forces the reader to read the test code (more on that in the readability section in this chapter). When you test just one concern, naming the test is easy. But wait, there’s more.

More disturbingly, in most unit test frameworks a failed assert throws a special type of exception that’s caught by the test framework runner. When the test framework catches that exception, it means the test has failed.

Most exceptions in most languages, by design, don’t let the code continue.

So, if this line fails the assert:

```
    expect(result).toBe(3);
```

This line will not execute at all:

```
    expect(callback).toHaveBeenCalledWith("I'm triggered");
```

The test method exits on the same line where the exception is thrown. Each of these can and should be considered different requirements, and they can and in this case likely should be implemented separately and incrementally one after the other.

Consider assert failures as symptoms of a disease. The more symptoms you can find, the easier the disease will be to diagnose. After a failure, subsequent asserts aren’t executed, and you miss seeing other possible symptoms that could provide valuable data (symptoms) that would help you narrow your focus and discover the underlying problem.

Let’s break it up into two separate tests:

```
describe("trigger", () => {
  it("triggers a given callback", () => {
    const callback = jest.fn();
    trigger(1, 2, callback);
    expect(callback).toHaveBeenCalledWith("I'm triggered");
  });

  it("sums up given values", () => {
    const result = trigger(1, 2, jest.fn());
    expect(result).toBe(3);
  });
});
```

Now we can clearly separate the concerns and each one can fail separately.

Checking multiple concerns in a single unit test adds complexity with little value. You should run additional concern checks in separate, self-contained unit tests so that you can see what really fails.

Sometimes it’s perfectly OK to assert multiple things in the same test, as long as they are not multiple concerns. Take this function and its associated test as an example. makePerson is design to build a new person object with some properties.

```
const makePerson = (x, y) => {
  return {
    name: x,
    age: y,
    type: "person",
  };
};

describe("makePerson", () => {
  it("createsperson given passed in values", () => {
    const result = makePerson("name", 1);
    expect(result.name).toBe("name");
    expect(result.age).toBe(1);
  });
});
```

In our test we are asserting on both name and age together, because they are part of the same concern (the person object’s building). If the first assert fails, we likely don’t care about the second assert because something might have gone terribly wrong building the person in the first place.

The test-break-up rule:

if the first assert fails, do you still care what the result of the next assert is?? If you do, you should probably separate the test into two tests.

7.8.6 Tests that keep changing

If your test is using the current date and time as part of its execution or assertions, then we can claim that every time the test runs, it’s a different test. The same can be said for tests that use random numbers, machine names, or anything that depends on grabbing a current value from outside the test's environment.

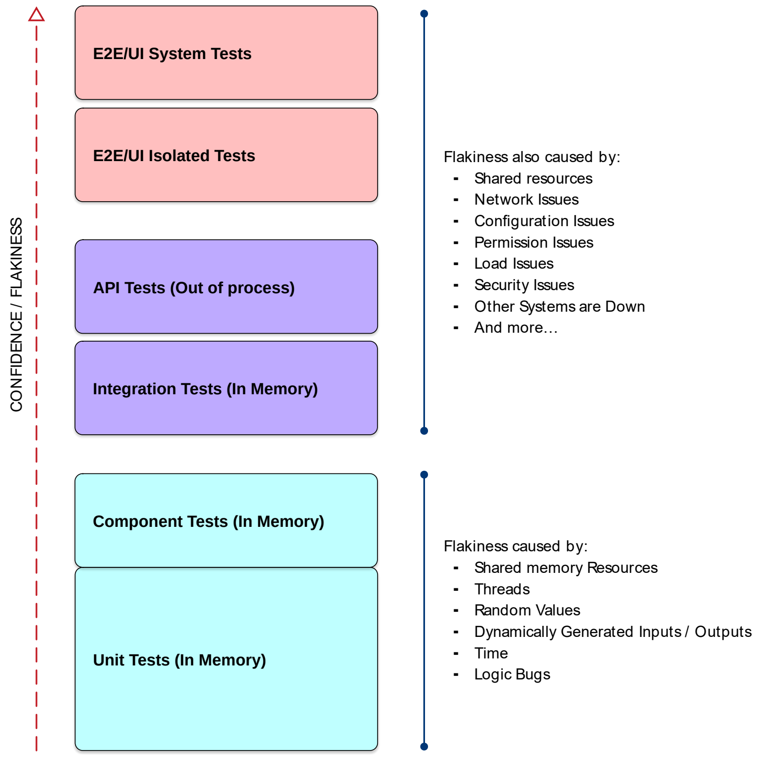
There's a big chance its results won't be consistent in their execution and results, and that means they can be flaky. "flaky" sucks for us as developers because it reduces our trust in the failed results of the test (as I discuss in the last section of this chapter).

Another huge potential issue with dynamically generated values is that if we don't know ahead of time what the input into the system might be, we also have to compute the expected *output* of the system, and that can lead to a buggy test which depends on repeating production logic as mentioned in section 7.4.

7.9 Dealing with Flaky Tests

I'm not sure who came up with the term "flaky tests" but it does fit the bill. It's used to describe tests that, given no changes to the code, will return inconsistent results. This might happen frequently or very little – but it does happen.

To understand where flakiness comes from, here's a diagram of the various levels of tests I consider as part of a testing strategy (We'll talk more about strategies in the 3rd part of this book):



The diagram is based on the amount of **real dependencies** that our tests have. Another way to think about it is in how many "moving parts" the tests have.

For this book, we're mostly concerning ourselves mostly with the bottom third of this diagram: Unit and Component tests, however, I just want to touch on the higher level flakiness to make sure I can give you some pointers on what to research.

At the lowest level, our tests (should) have full control over all of their dependencies (no moving parts), either by faking them or because they run purely in memory and can be configured (much like we did in the chapters about mocks and stubs).

Execution paths in the code are fully deterministic because all the initial states and expected return values from various dependencies have been pre-determined. The code path is almost "static" in nature – if it does return the wrong expected result then truly something important might have changed in the production code's execution path control flow or logic.

As we go up the levels, our tests "shed away" more and more stubs and mocks and start using more and more "real" dependencies such as databases, networks, configuration, and more. This in turn means more "moving parts" that we have less control over, and might change our execution path, return unexpected values or fail to execute at all.

At the highest level, there are no "fake" dependencies. Everything our tests rely on is real: including any 3rd party services, security and network layers, and configuration. These types of tests usually require us to setup up an environment that is as close to production scenarios as possible, if not running right on production environments.

The higher up we go in the test diagram, we **should** get higher confidence that our code works, unless we don't trust the tests' results. Unfortunately, the higher up we go in the diagram, the more chances we have for our tests to become flaky because of how many moving parts there are.

We might assume that tests at the lowest level shouldn't have any flakiness issues because there shouldn't be any moving parts that cause any flakiness. Theoretically true, but in reality people do still manage to shove moving parts into lower level tests and cause them to be flaky such as using the current date and time, machine name, network, file system and more.

In terms of obviousness – higher level tests are much more likely to be flaky because of how many real dependencies they use that they do not control. Lower level tests can often suffer from flakiness as well but for less obvious reasons.

7.9.1 Detecting Flakiness:

A test fails sometimes, without us touching production code. For example:

1. A test fails every 3rd run
2. A test fails once every unknown amount of times
3. A test fails when various external conditions fail such as network, database availability, other APIs not being available, environment configuration and more.

7.9.2 Flaky tests take more time to run and understand.

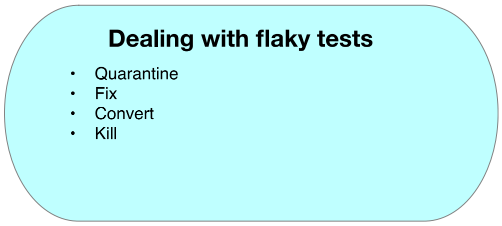
To add to that salad of pain, each dependency the test uses (network, file system, threads, etc…) usually adds time to the test run. Network calls take time database access takes time. Waiting for threads to finish takes time. Reading configuration takes time. Waiting for asynchronous tasks takes time.

It also takes longer to figure out why a test is failing because debugging them or reading through huge amounts of logs is heartbreakingly time-consuming and will drain your soul slowly into the abyss of "time to update my resume" -land.

7.9.3 What can you do once you've found a flaky test?

It's important to realize that flaky tests can cost a lot to the organization. You should aim to have zero flaky tests as a long-term goal. Here are some ways to reduce the costs associated with handling flaky tests:

1. Agree on what flaky means to your organization. Example: Run your test suite 10 times without any production code changes, and count all the tests that were NOT consistent in their results (i.e did not fail 10 times or did not pass 10 times).
2. Place any test deemed "flaky" in a special category or folder of tests that can be run separately.
3. Prune and remove flakiness from quarantined tests.



Dealing with flaky tests

- Quarantine
- Fix
- Convert
- Kill

a) **Quarantine**: I recommend to remove all flaky tests from the regular delivery build so they do not create noise and quarantine them to their own little pipeline temporarily. Then, one by one go over each of the flaky tests and play my favorite flaky game: "fix, convert or kill".

b) **Fix**: Make the test not flaky if possible - by controlling its dependencies. For example if it requires data in the database, insert the data into the database as part of the test. Sometimes fixing a test won't be possible - that's when converting the test might make sense.

c) **Convert**: Remove flakiness by converting the test into a lower level test - by removing and controlling one or more of its dependencies. For example, simulate a network endpoint instead of using a real one.

d) **Kill**: Seriously consider whether the value the test brings is large enough to continue to run it and pay the noise costs it creates – sometimes old flaky tests are better off dead and buried. Sometimes they are already covered by newer, better tests – and the old tests are pure technical debt that we can get rid of. Sadly, many engineering managers are reluctant remove these old tests because of the "sunken cost fallacy" – there was so much effort put into them it would be a waste to delete them – however, at this point it might cost you more to keep the test than to delete it – so I recommend seriously considering this option for many of your flaky tests.

7.9.4 Preventing flakiness in higher level tests

If you're interested in preventing flakiness in higher level tests – your best bet is to make sure that your tests are repeatable on any environment after any deployment. That could mean:

1. Roll back any changes they've made to external shared resources
2. Do not depend on other tests changing external state
3. Gain some control over external systems and dependencies by either:
    a) Being able to recreate them at will (Google "infrastructure as code")
    b) Create dummies of them that you can control
    c) Create special test accounts on them and pray that they stay safe.

This last point, external dependencies control, can be quite difficult to impossible when using some external systems managed by other companies. When that's true it's worth considering:

1. Removing some of the higher-level tests if some low-level tests already cover those scenarios
2. Converting some of the higher-level tests to a set of lower level tests
3. If you're writing new tests, consider a pipeline-friendly testing strategy with test-recipes, such as the one I explain in the 3rd part of this book in chapter 10.

7.10 Summary

In this chapter we covered the idea of trust. The most important thing to take form this chapter is that if you don't trust a test when it's failing, you might ignore a real bug, and if you don't trust a test when it's passing, you'll end up doing lots of manual debugging and testing. Both of these outcomes are supposed to be reduced by having good tests, but if we don't reduce them, *and* we spend all this time writing these tests that we don't trust – what's the point in writing them in the first place?

8 Maintainability

This chapter covers

- **Root causes for failing tests**
- **Common avoidable changes to test code**
- **Improving maintainability of tests that aren't currently failing**

Tests become a real pain if we keep having to change existing tests, on top of the changes we're adding to production code. In this chapter we'll focus on the maintainability aspect of tests.

This can manifest in our needing to change existing tests too much, or for no good reason. If we could avoid changing existing tests when we change production code, we can start to have hope that our tests are helping rather than hurting our bottom line.

Tests can enable us to develop faster, unless they make us go slower (due to all the changes needed) - and that's what we're going to tryu and work through here.

Unmaintainable tests can ruin project schedules and are often "put aside" when the project is put on a more aggressive schedule. Developers will simply stop maintaining and fixing tests that take too long to change or that need to change very often on very minor production code changes.

This chapter presents a series of practices related to the pillar of maintainability that you can use when doing test reviews.

8.1 Root causes for test changes

If maintainability is a measure of how often we are forced to change the tests, we'd like to minimize the number of times that happens. This forces us to ask these questions if we ever want to get down to the root causes:

- When do we notice that a test might require a change? (When tests fail)
- Why do tests fail?

- Which test failures "force" us to change the test?
- When do we "choose" to change a test even if we are not forced to?

8.2 Forced changes by failing tests

A failing test is usually the first sign of potential trouble for our maintainability aspect. Of course, it could be that we've found a real bug in production code, but when that's not the case, what other reasons do tests have to fail?

If we ever wanted to measure test maintainability, we can start by measuring amount of test failures, and the reason for each failure, over time.

A test can suddenly fail for several reasons.

8.2.1 Breaking when there's a production bug (great!)

A production bug occurs when you change the production code and an existing test breaks. If indeed this is a bug in the code under test, your test is fine, and you shouldn't need to touch the test (unless you'd like to refactor it further as we'll see later). This is the best and most desired outcome of having tests.

8.2.2 Breaking when a test has a bug (increase trust)

We covered test bugs in the previous chapter about trustworthy tests – obviously we'd like to avoid this as much as possible. See chapter 7 for more details on this topic.

8.2.3 Breaking when a test is not relevant, or conflicts with another test (remove)

A conflict arises when either of the following occurs:

- the production code introduces a new feature that's in direct conflict with one or more existing tests. This means that instead of the test discovering a bug, it discovers conflicting or new requirements.
- In a test driven cycle, a new test is written to reflect a new requirement, and as we make it pass by adding the new requirement, an old test starts failing, that represents an old expectation on how the production code should work.

Either way, it means that either the existing failing test is no longer relevant, or that out new requirement is wrong. Assuming the requirement is correct, we could probably go ahead and delete the no-long-relevant test.

Note: A common exception to the "remove the test" rule is when you're working with *feature toggles*. We'll touch on feature toggles in chapter 10 as part of the testing strategies discussion.

8.2.4 Breaking when the Semantics or API in our production code change

This falls in the bucket of "let's avoid this as much as possible".

A test can fail when the production code under test changes so that a function or object being tested now needs to be used differently, even though it may still have the same end functionality.

Consider the following `PasswordVerifier` class that required two constructor parameters:

- An array of `rules` (each is a function that takes an input and returns a boolean)
- An `ILogger` interface.

You'll see other maintainability techniques later in this chapter.

Listing 8.1: A Password verifier with 2 constructor parameters

```
ch-8-maintain/00-password-verifier.ts

import {ILogger} from "./interfaces/logger";

export class PasswordVerifier {
    ...
    constructor(rules: ((input) => boolean)[], logger: ILogger) {
        this._rules = rules;
        this._logger = logger;
    }

    ...
}
```

We could write a couple of tests that look like this for this

Listing 8.2: tests without factory functions

```
ch-8-maintain/00-password-verifier.v1.spec.ts:

import { PasswordVerifier } from "./00-password-verifier";

describe("password verifier 1", () => {
  it("passes with zero rules", () => {
    const verifier = new PasswordVerifier([], { info: jest.fn() });
    const result = verifier.verify("any input");
    expect(result).toBe(true);
  });

  it("fails with single failing rule", () => {
    const failingRule = (input) => false;
    const verifier =
   new PasswordVerifier([failingRule], { info: jest.fn() });
    const result = verifier.verify("any input");
    expect(result).toBe(false);
  });
});

```

As we look at these tests from a maintainability point of view, we can think of several potential changes we will likely need to make in the future that can be minimized. Keep in mind when I say "future" I mean this:

CODE USUALLY LIVES FOR A LONG TIME.

Consider that the code you’re writing will live in the codebase for at least 4-6 years and sometimes a decade. Over that time, what is the likelihood that the design of `PasswordVerifier` will change? Even simple things like the constructor accepting a couple of more parameters, or the parameter types changing become more likely over a longer timeframe.

So let’s write down a few of the possible changes that could happen in the future:

- Constructor for `PasswordVerifier` gets a new parameter or removes one parameter
- One of the parameters for `PasswordVerifier` changes to a different type
- The amount of `ILogger` functions or their signature changes over time
- The usage pattern changes so you don’t need to `new` up a new `PasswordVerifier`, but just use the functions in it directly.

If any of these things happen, how many tests will we need to change?

Right now we’d need to change *all the tests that instantiate* `PasswordVerifier`.

Could we prevent the need for some of these changes?

Let’s pretend the future is here, and someone changes out production code, and our test fear have come true - The class under test has changed constructor signature to use `IComplicatedLogger` instead of `ILogger`:

Listing 8.3: A breaking change in a constructor

```
ch-8-maintain/00-password-verifier2.ts

export class PasswordVerifier2 {
  private _rules: ((input: string) => boolean)[];
  private _logger: IComplicatedLogger;

  constructor(rules: ((input) => boolean)[],
                    logger: IComplicatedLogger) {
    this._rules = rules;
    this._logger = logger;
  }
...
}
```

As it stands, we would have to change any tests that directly instantiates `PasswordVerifier`.

FACTORY FUNCTIONS DECOUPLE CREATION OF OBJECT UNDER TEST

A simple method to avoid this pain in the future is to decouple or abstract away the creation of the code under test so that the constructure changes only need to be dealt with in a centralized location. A function who's sole purpose is to create and preconfigure an instance of an object is usually called a "factory" function or method. A more advanced version of this (which we won't cover here) is called the "Object Mother" pattern.

Factory functions can help us mitigate this issue. The next two listings show:

- Listing 8.4: How we could have initially written the tests before the signature change
- Listing 8.5: How we easily adapt to the signature change

We’ve extracted the creation of `PasswordVerifier` into its own centralized factory function. We’ve done the same for the `fakeLogger`. When any of the issues mentioned before happens in the future, we’ll only need to change out factory functions, and the tests will usually not need to be touched.

Listing 8.4: Refactoring to factory functions (before)

```
ch-8-maintain/00-password-verifier.v2.spec.ts
import { PasswordVerifier } from "./00-password-verifier";
import { ILogger } from "./interfaces/logger";

describe("password verifier 1", () => {
  const makeFakeLogger = () => {
    return { info: jest.fn() }; #A
  };

  const makePasswordVerifier = (
    rules: ((input) => boolean)[],
    fakeLogger: ILogger = makeFakeLogger()
  ) => {
    return new PasswordVerifier(rules, fakeLogger); #B
  };

  it("passes with zero rules", () => {
    const verifier = makePasswordVerifier([]); #C

    const result = verifier.verify("any input");

    expect(result).toBe(true);
  });

      ...
```

#A: A centralized point for creating a fake logger
#B: A centralized point for creating a PasswordVerifier
#C, #D: Using factory functions to create `PasswordVerifier`

In the next listing – 8.5, we’ve refactored the tests based on the signature change. Notice the change doesn’t involve changing the tests, but only the factory functions. That’s the type of manageable change I can live with in a real project.

Listing 8.5: Refactoring factory methods to fit new signature

```
ch-8-maintain/00-password-verifier.v3.spec.ts

import { IComplicatedLogger } from "./interfaces/complicated-logger";
import { Substitute } from "@fluffy-spoon/substitute";
import { PasswordVerifier2 } from "./00-password-verifier2";

describe("password verifier (ctor change)", () => {
  const makeFakeLogger = () => {
    return Substitute.for<IComplicatedLogger>();
  };

  const makePasswordVerifier = (
          rules: ((input) => boolean)[],
                    fakeLogger: ICompLicatedLogger = makeFakeLogger()
 ) => {
    return new PasswordVerifier2(rules, fakeLogger);
  };

 ... //the tests
  });
});
```

8.2.5 Breaking when some other tests or test code changes (decrease test coupling)

A lack of test isolation is a huge cause of test blockage I've seen while consulting and working on unit tests. The basic concept to keep in mind is that a test should always run in its own little world, isolated from even the knowledge that other tests out there may do similar or different things.

Sidebar: The test that cried "fail"

One project I was involved in had unit tests behaving strangely, and they got even stranger as time went on. A test would fail and then suddenly pass for a couple of days straight. A day later, it would fail, seemingly randomly, and other times it would pass even if code was changed to remove or change its behavior. It got to the point where developers would tell each other, "Ah, it's OK. If it sometimes passes, that means it passes."

Properly investigated, it turned out that the test was calling out a different (and flakey) test as part of its code, and when the other test failed , it would break the first test.

It took us three days to untangle the mess, after spending a month trying various workarounds for the situation. When we finally had the test working correctly, we discovered that we had a bunch of real bugs in our code that we were ignoring because the test had its own bugs and issues. The story of the boy who cried "wolf" holds true even in development.

When tests aren't isolated well, they can step on each other's toes enough to make you miserable, making you regret deciding to try unit testing on the project and promising yourself never again. I've seen this happen. Developers don't bother looking for problems in the tests, so when there's a problem with them, it can take a lot of time to find out what's wrong.

The easiest symptom is what I call "constrained test order".

CONSTRAINED TEST ORDER

In short, a constrained test order happens when a test assumes that a previous test executed first, or did not execute first, because it relies on some shared state that is setup or reset by the other test.

The longer version:

If one test changes a shared variable in memory or some external resource like a database, and another test depends on that variable's value after the first tests' execution, we have a dependency between the tests based on order. Couple that with the fact that most test runners don't (and won't, and maybe shouldn't!) guarantee that tests will run in a specific order. That means that if you ran all your tests today, and all your tests in one week with a new version of the test runner, The tests may not run in the same order the did based on each runner's internal implementation.

To illustrate the problem let's look at simple scenario. Figure 8.1 shows a `SpecialApp` object that uses a `UserCache` object. The user cache holds a single shared instance (a singleton) that is shared as a caching mechanism for the application, and incidentally, also for the tests.

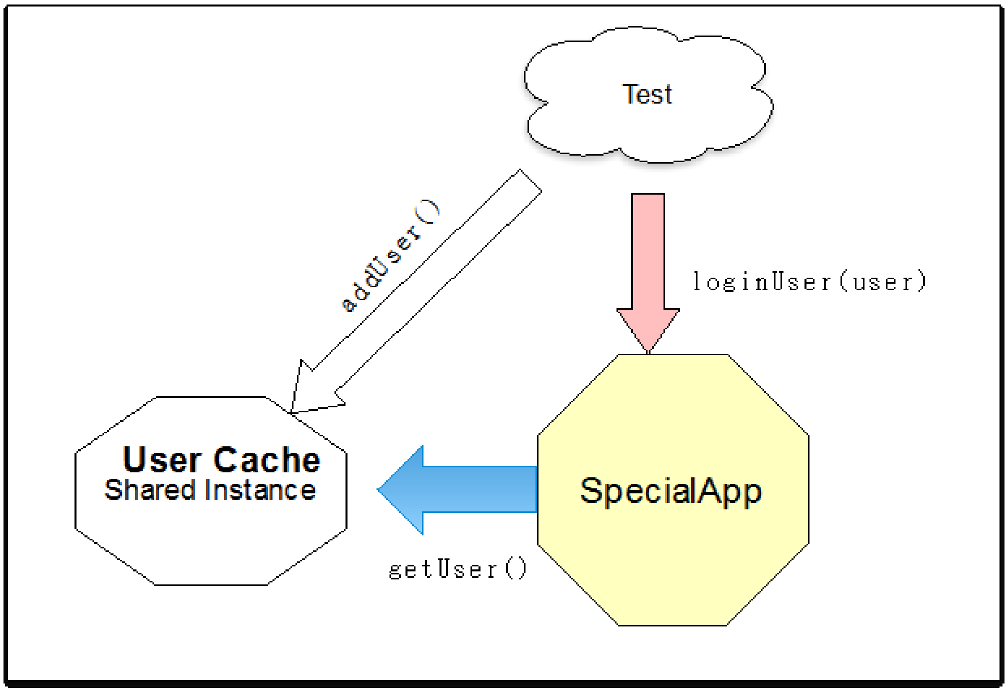


Figure 8.1: A shared user cache instance

Here's the implementation of `SpecialApp` and the user cache and the `IUserDetails` interface:

Listing 8.6: shared user cache and associated interfaces

```
ch-8-maintain/sharedUserCache.ts
   export interface IUserDetails {
     key: string;
```

```
  password: string;
}

export interface IUserCache {
  addUser(user: IUserDetails): void;
  getUser(key: string);
  reset(): void;
}

export class UserCache implements IUserCache {
  users: object = {};
  addUser(user: IUserDetails): void {
    if (this.users[user.key] !== undefined) {
      throw new Error("user already exists");
    }
    this.users[user.key] = user;
  }

  getUser(key: string) {
    return this.users[key];
  }

  reset(): void {
    this.users = {};
  }
}

let _cache: IUserCache;
export function getUserCache() {
  if (_cache === undefined) {
    _cache = new UserCache();
  }
  return _cache;
}

```

And here's the `SpecialApp` implementation:

Listing 8.7: SpecialApp implementation

```
ch-8-maintain/specialApp.ts
import { getUserCache, IUserCache, IUserDetails } from "./sharedUserCache";

export class SpecialApp {
  loginUser(key: string, pass: string): boolean {
    const cache: IUserCache = getUserCache();
    const foundUser: IUserDetails = cache.getUser(key);
    if (foundUser?.password === pass) {
      return true;
    }
    return false;
  }
}
```

It's a very simplistic implementation because the purpose is to show something in the tests for this application and the user cache. So, don't worry about the `SpecialApp` too much. It's just an exercise. Let's look at the tests:

Listing 8.8: tests that need to run in a specific order

```
ch-8-maintain/specialApp.spec.ts
import { getUserCache } from "./sharedUserCache";
import { SpecialApp } from "./specialApp";

describe("Test Dependence", () => {
  describe("loginUser with loggedInUser", () => {
    test("A: no user - login fails", () => {
      const app = new SpecialApp();
      const result = app.loginUser("a", "abc");
      expect(result).toBe(false);
    });

    test("B: can only cache each user once", () => {
      getUserCache().addUser({
        key: "a",
        password: "abc",
      });

      expect(() =>
        getUserCache().addUser({
          key: "a",
          password: "abc",
        })
      ).toThrowError("already exists");
    });

    test("C: user exists, login succeeeds () => {
      const app = new SpecialApp();
      const result = app.loginUser("a", "abc");
      expect(result).toBe(true);
    });
  });
});

```

Notice that:

- Test “A” requires that test “B” did not execute yet, because a user is added to the cache only in test “B”
- Test “C” required that test “B” has already executed since it requires a user to already exist in the cache.

If, for example, we’d run only test C using jest’s `it.only` keyword, it would fail:

```
test.only("C: user exists, login succeeeds () => {
   const app = new SpecialApp();
   const result = app.loginUser("a", "abc");
   expect(result).toBe(true);  /// ← wOULD FAIL
 });
```

This type of anti-pattern usually happens when we try to “reuse” parts of the tests without extracting helper functions. We end up expecting a different test to run before, “saving us” from doing some of the set up. Initially this works… until it doesn’t.

Refactoring this we can take a few steps:

- Extract a helper function for adding a user
- Reusing this function form multiple tests
- Reset the user cache between tests.

Here’s how we can refactor the tests to avoid this problem:

Listing 8.:9 Refactored tests to remove order dependence

```
ch-8-maintain/specialApp.v2.spec.ts
import { getUserCache } from "./sharedUserCache";
import { SpecialApp } from "./specialApp";

const addDefaultUser = () => //#A
  getUserCache().addUser({
    key: "a",
    password: "abc",
  });
const makeSpecialApp = () => new SpecialApp(); //#B

describe("Test Dependence v2", () => {
  beforeEach(() => getUserCache().reset()); //#C
  describe("user cache", () => {
    test("B: can only add cache use once", () => {
      addDefaultUser();                       //#D

      expect(() => addDefaultUser())
           .toThrowError("already exists");
    });
  });

  describe("loginUser with loggedInUser", () => {
    test("A: user exists, login succeeds", () => {
      addDefaultUser();                       //#D
      const app = makeSpecialApp();

      const result = app.loginUser("a", "abc");
      expect(result).toBe(true);
    });

    test("C: user missing, login fails", () => {
      const app = makeSpecialApp();

      const result = app.loginUser("a", "abc");
      expect(result).toBe(false);
    });
  });
});
```

#A: Extracted user creation helper function
#B: Extracted factory function
#C Reset user cache between tests
#D: call reusable helper functions

There are many things going on in here so let’s list them:

- First we extracted two helper functions: a factory function `makeSpecialApp` and a helper `addDefaultUser` function that we can reuse.

- Next we created a very important `forEach` function that resets the user cache before each test. Whenever we have a shared resource like that, I almost always have a `beforeEach` or `afterEach` function that resets it to its original condition before or after the test ran.
- Tests `A` and `C` now run in their own little nested `describe` structure. They also both use the factory function `makeSpecialApp` and one of them is using `addDefaultUser` to make sure it does not require any other test to run first.
- Test `B` runs in its own nested `describe` and reuses the `makeDefaultUser` function.

8.3 Refactoring to increase maintainability

Up until now I've discussed test failures which force us to make changes. Let's discuss changes that we *choose* to do to make tests easier to maintain over time.

8.3.1 Avoid Testing private or protected methods

This section would apply more for object oriented languages, as well as TypeScript based JavaScript. Private or protected methods are usually private for a good reason in the developer's mind. Sometimes it's to hide implementation details, so that the implementation can change later without the end functionality changing. It could also be for security-related or IP-related reasons (obfuscation, for example).

When you test a private method, you're testing against a contract internal to the system, which may well change. Internal contracts are dynamic, and they can change when you refactor the system. When they change, your test could fail because some internal work is being done differently, even though the overall functionality of the system remains the same.

For testing purposes, the public contract (the overall functionality) is all that you need to care about. Testing the functionality of private methods may lead to breaking tests, even though the overall functionality is correct.

Think of it this way: no private method exists in a vaccum. Somewhere down the line ,something has to call it, or it will never get triggered. Usually there's a public method that ends up invoking this private one, and if not, there's always a public method up the chain of calls that gets invoked.

That means that any private method is always part of a bigger unit of work, or a use case in the system, that starts out with a public API and ends with one of the three end results: return value, state change, or third-party call (or all three).

So, **if you see a private method, find the public use case in the system that will exercise it**. If you test only the private method and it works, that doesn't mean that the rest of the system is using this private method correctly or handles the results it provides correctly. You might have a system that works perfectly on the inside, but all that nice inside stuff is used horribly wrong from the public APIs.

Sometimes if a private method is worth testing, it might be worth making it public, static, or at least internal and defining a public contract against any code that uses it. In some cases, the design may be cleaner if you put the method in a different class altogether. We'll look at these approaches in a moment.

Does this mean there should eventually be no private methods in the code base? No. With TDD, you usually write tests against methods that are public, and those public methods are later refactored into calling smaller, private methods. All the while, the tests against the public methods continue to pass.

MAKING METHODS PUBLIC

Making a method public isn't necessarily a bad thing. In a more functional world it's not even an issue. It may also seem to go against the object-oriented principles many of us were raised on.

Consider that wanting to test a method could mean that the method has a known behavior or contract against the calling code. By making it public, you're making this official. By keeping the method private, you tell all the developers who come after you that they can change the implementation of the method without worrying about unknown code that uses it.

EXTRACTING METHODS TO NEW CLASSES OR MODULES

If your method contains a lot of logic that can stand on its own, or it uses specialized state variables in the class or module that's relevant only to the method in question, it may be a good idea to extract the method into a new class or its own module, with a specific role in the system. You can then test that class separately. Michael Feathers' *Working Effectively with Legacy Code*, has some good examples of this technique, and *Clean Code* by Robert Martin can help with figuring out when this is a good idea.

MAKING STATELESS PRIVATE METHODS PUBLIC STATIC

In languages that support this feature, if your method is completely stateless, some people choose to refactor the method by making it static. That makes it much more testable but also states that this method is a sort of utility method that has a known public contract specified by its name.

8.3.2 Remove duplication

Duplication in your unit tests can hurt you as developers just as much as (if not more than) duplication in production code. This is because any change in one piece of code that has duplicates, will force you to change all the duplicates as well. When we're dealing with tests, there's more risk of you just avoiding this trouble and actually deleting or ignoring tests, instead of fixing them.

The DRY principle should be in effect in test code the same as in production code. Duplicated code means more code to change when one aspect you test against changes. Changing a constructor or changing the semantics of using a class can have a major effect on tests that have a lot of duplicated code.

Using helper functions as we've seen in previous examples in this chapter can help to reduce duplication in out tests.

NOTE: Removing duplication can also go too far and hurt readability. We'll talk about that in the next chapter on readability.

8.3.3 Avoid setup methods(in a maintainable manner)

I am not a fan of the `beforeEach` function (also called a *setup* function) that happens once before each test has often been used to remove duplication. I much prefer the **helper function** way.

Setup functions are too easy to abuse—enough so that developers tend to use it for things it wasn't meant for, and tests become less readable and maintainable as a result.

Many developers abuse setup methods in several ways:

- Initializing objects in the setup method that are used in only some tests in the file
- Having setup code that's lengthy and hard to understand
- Setting up mocks and fake objects within the setup method

Also, setup methods have limitations, which you can get around using simple helper methods:

- Setup methods can only help when you need to initialize things.
- Setup methods aren't always the best candidates for duplication removal. Removing duplication isn't always about creating and initializing new instances of objects. Sometimes it's about removing duplication in assertion logic, calling out code in a specific way.
- Setup methods can't have parameters or return values.
- Setup methods can't be used as factory methods that return values. They're run before the test executes, so they must be more generic in the way they work. Tests sometimes need to request specific things or call shared code with a parameter for the specific test (for example, retrieve an object and set its property to a specific value).
- Setup methods should only contain code that applies to all the tests in the current test class, or the method will be harder to read and understand.

I've stopped using setup methods for tests I write for 99% of the time. Test code should be nice and clean, just like production code. But if your production code looks horrible, please don't use that as a crutch to write unreadable tests. Just use factory and helper methods, and make the world a better place for the generation of developers that has to maintain your code in 5 or 10 years.

An example of moving from `beforeEach` to helper functions is shown in chapter 2, in section 2.71

8.3.4 Using parameterized tests to remove duplication

Another great option to replace setup methods if all your tests look the same is to use parameterized tests.

Different test frameworks in different languages support parametrized tests.If you're using jest, you can use the built in `test.each or it.each` function as show in the following listing:

Listing 8.10: parametrized tests with jest

```
ch-8-maintain/parametrized.spec.ts

01    const sum = numbers => {
02        if (numbers.length > 0) {
03            return parseInt(numbers);
04        }
05        return 0;
06    };
07
08    describe('sum with regular tests', () => {
09        test('sum number 1', () => {
10            const result = sum('1');
11            expect(result).toBe(1);
12        });
13        test('sum number 2', () => {
14            const result = sum('2');
15            expect(result).toBe(2);
16        });
17    });
18
19    describe('sum with parametrized tests', () => {
20        test.each([
21            ['1', 1],
22            ['2', 2]
23        ])('add ,for %s,returns that number', (input, expected) => {
24                const result = sum(input);
25                expect(result).toBe(expected);
26            }
27        )
28    });
```

In the first `describe` block we show two tests that repeat each other with different input values and expected outputs. In the second `describe` block we're using `test.each` to provide an array of arrays, each sub array lists all the values needed for the test function.

- In line 23 we're providing two parameters to the test function that will be mapped to the arrays we provided in lines 21 and 22.
- Note that in line 23 we're actually invoking a new function with the braces opening just before the 'add…' string. The syntax is a bit annoying but nothing we can't get used to.

Parameterized tests can help reduce a lot of duplication between tests, but we should also be careful to only use this technique in cases where we are repeated the exact same scenario (the middle part in my naming convention) and only the input and output changes. More about that in the readability chapter.

8.4 Avoid overspecification

An over specified test is one that contains assumptions about how a specific unit under test (production code) should implement its internal behavior, instead of only checking that the end behavior (exit points) is correct.

Here are ways unit tests are often over specified:

- A test asserts purely internal state in an object under test.
- A test uses multiple mocks.
- A test uses stubs also as mocks.
- A test assumes specific order or exact string matches when it isn't required.

Tip

This topic is also discussed in xUnit Test Patterns: Refactoring Test Code by Gerard Meszaros.

Let's look at some examples of over specified tests.

8.4.1 Internal behavior overspecification with mocks

A very common any pattern is to verify that some internal function in a class or module is called, instead of checking the exit point of the unit of work. Here's a password verifier that calls an internal function, which the test shouldn't care about.

Listing 8.11:Production code that calls a protected function

```
ch-8-maintain/00-password-verifier4.ts

import { IComplicatedLogger } from "./interfaces/complicated-logger";

export class PasswordVerifier4 {
  private _rules: ((input: string) => boolean)[];
  private _logger: IComplicatedLogger;

  constructor(rules: ((input) => boolean)[],
                      logger: IComplicatedLogger) {
    this._rules = rules;
    this._logger = logger;
  }

  verify(input: string): boolean {
    const failed = this.findFailedRules(input);

    if (failed.length === 0) {
      this._logger.info("PASSED");
      return true;
    }
    this._logger.info("FAIL");
    return false;
  }

  protected findFailedRules(input: string) {
    const failed = this._rules
      .map((rule) => rule(input))
      .filter((result) => result === false);
    return failed;
  }
}
```

Notice that we're calling the protected `findFailedRules()` function to get a result from it, and do a calculation on the result.

And here's out test:

Listing 8.12: over specified test verifying call to protected function

```
ch-8-maintain/overspec.1.spec.ts

...
05    describe("verifier 4", () => {
06      describe("overcpecify protected function call", () => {
07        test("checkfailedFules is called", () => {
08          const pv4 = new PasswordVerifier4(
09            [],Substitute.for<IComplicatedLogger>()
11          );
12          // the fake function returns an empty array
13          const failedMock = jest.fn(() => []);
14          pv4["findFailedRules"] = failedMock;
15
16          pv4.verify("abc");
17
18          //Don't do this
19          expect(failedMock).toHaveBeenCalled();
20        });
21      });
22    });
23
```

The anti pattern here manifests in line 19. We're proving something that isn't an exit point. We're proving that the code calls some internal function, but what does that prove really? We're not checking the calculation was correct on the result, we're simply testing for the sake of testing.

The hint is: if the function is returning a value , usually it's not the thing we're supposed to be using a mock object for because it does not represent the exit point.

The exit point is the value returned from the verify function. We shouldn't care that the internal function even exists.

By verifying against a mock of a protected function that is not an exit point we are coupling our test implementation into the internal implementation of the code under test, for not real benefit. When the internal calls change (and they will) we will have to change all the tests associated with these calls as well, and that will not be a positive experience, guaranteed.

WHAT SHOULD WE DO INSTEAD?

Look for the exit point. The real exit point depends on the type of test we wish to perform:

- For a value based test, which I would highly recommend to lean towards when possible, we look for a return value from the called function. In this case the `verify()` function returns a value so it's the perfect candidate for a value based test:

```
pv4.verify("abc");
```

- For a state based test we look for a sibling function (a function that exists at the same level of scope as the entry point. For example, a person's firstname() and lastname() can be considered sibling functions) .or sibling property that is affected by calling the `verify()` function. That is where we should be asserting.

  In this codebase there isn't anything affected by calling `verify()` that is visible at the same level, so it is not a good candidate for state based testing.

- For a 3d party test we would have to use a mock, and that would require us to find out where is the "fire and forget" location inside the code. The `findFailedRules` function isn't that because it is actually delivering information back into out `verify()` function. It isn't a good candidate and in this case there's no real 3rd party dependency that we have to take over.

8.4.2 Exact outputs and ordering overspecification

A common anti pattern I see is when a collection of values are returned, the test over specifies the order and the schema (structure) of the returned values = two things that we might not care about, but implicitly take on the burden of changing the test when they do change.

Here's an example of a `verify()` function that takes on multiple inputs and returns a list of result objects:

Listing 8.13: a verifier that returns a list of outputs

```
ch-8-maintain/00-password-verifier5.ts

interface IResult {
  result: boolean;
  input: string;
}

export class PasswordVerifier5 {
  private _rules: ((input: string) => boolean)[];

  constructor(rules: ((input) => boolean)[]) {
    this._rules = rules;
  }

  verify(inputs: string[]): IResult[] {
    const failedResults =
   inputs.map((input) => this.checkSingleInput(input));
    return failedResults;
  }

  private checkSingleInput(input: string) {
    const failed = this.findFailedRules(input);
    return {
      input,
      result: failed.length === 0,
    };
  }
...
```

Our function returns a list of `IResult` objects with the `input`, and the `result` for each. Here's a test that makes an implicit check on both the ordering of the results and the structure of each result, on top of checking the value of the results:

Listing 8.14: overcpecify order and structure (schema)

```
ch-8-maintain/overspec.2.spec.ts

test("overspecify order and schema", () => {
  const pv5 =
   new PasswordVerifier5([(input) => input.includes("abc")]);
  const results = pv5.verify(["a", "ab", "abc", "abcd"]);
  expect(results).toEqual([
    { input: "a", result: false },
    { input: "ab", result: false },
    { input: "abc", result: true },
    { input: "abcd", result: true },
  ]);
});
```

How many reasons does this test have to require a change in the future? Here are a few:

When the *length of the results changes*

When each result object gains or removes a property (even if the test doesn't' care about other properties)

When the order of the results changes (even if it might not be important for the current test)

If any of these changes might happen in the future, but your test is just focused don checking the logic that uses out custom rules for the verifier, there's going to be a lot of pain involved for the poor souls who will have to maintain out tests.

We can reduce some of that pain by writing the test like this:

Listing 8.15: ignore schema of results

```
ch-8-maintain/overspec.2.spec.ts

est("overspecify order but ignore schema", () => {
  const pv5 =
    new PasswordVerifier5([(input) => input.includes("abc")]);
  const results = pv5.verify(["a", "ab", "abc", "abcd"]);
  expect(results.length).toBe(4);
  expect(results[0].result).toBe(false);
  expect(results[1].result).toBe(false);
  expect(results[2].result).toBe(true);
  expect(results[3].result).toBe(true);
});
```

Instead of providing the full expected output, we can simple assert on the values of specific properties in the output. We're still stuck if the order of the results changes. If we don't care about the order we can simple check if the output contains a specific result, or in this case we can "count" how many false and true results we've gotten:

Listing 8.16: ignoring order and schema

```
ch-8-maintain/overspec.2.spec.ts

test("ignore order and schema", () => {
  const pv5 =
   new PasswordVerifier5([(input) => input.includes("abc")]);

  const results = pv5.verify(["a", "ab", "abc", "abcd"]);

  const falseResults = results.filter((res) => !res.result);
  const trueResults = results.filter((res) => res.result);
  expect(results.length).toBe(4);
  expect(falseResults.length).toBe(2);
  expect(trueResults.length).toBe(2);
});
```

Now the order of the results can change, or they can have extra values added but out test will only fail if the calculation of the true/false results changes.

It's not perfect, I don't like doing some of that filtering action in the test but it's simple enough that I might accept it in this specific case.

Overspecifying Strings

Another common pattern people tend to repeat is to have asserts against hardcoded strings in the unit's return value or properties, when only a specific part of a string is necessary. Ask yourself, "Can I check if a string *contains* something rather than *equals* something?"

Here's a password verifier that gives us a message describing how many rules were broken during a verification:

Listing 8.17: a verifier that returns a string message

```
ch-8-maintain/00-password-verifier6.ts

export class PasswordVerifier6 {
  private _rules: ((input: string) => boolean)[];
  private _msg: string = "";

  constructor(rules: ((input) => boolean)[]) {
    this._rules = rules;
  }

  getMsg(): string {
    return this._msg;
  }

  verify(inputs: string[]): IResult[] {
    const allResults =
   inputs.map((input) => this.checkSingleInput(input));
    this.setDescription(allResults);
    return allResults;
  }

  private setDescription(results: IResult[]) {
    const failed = results.filter((res) => !res.result);
    this._msg = `you have ${failed.length} failed rules.`;
    }
     ...
```

Listing 8.18 shows two tests that use `getMsg()`.

Listing 8.18: over specifying a string using equality

```
ch-8-maintain/overspec.3.spec.ts

import { PasswordVerifier6 } from "./00-password-verifier6";

describe("verifier 6", () => {
  test("over specify string", () => {
    const pv5 =
   new PasswordVerifier6([(input) => input.includes("abc")]);

    pv5.verify(["a", "ab", "abc", "abcd"]);
    const msg = pv5.getMsg();
    expect(msg).toBe("you have 2 failed rules.");
  });

//Here's a better way to write this test
  test("more future proof string checking", () => {
    const pv5 =
   new PasswordVerifier6([(input) => input.includes("abc")]);

    pv5.verify(["a", "ab", "abc", "abcd"]);
    const msg = pv5.getMsg();
    expect(msg).toMatch(/2 failed/);
  });
});
```

The first test checks that the string equals exactly another string. This backfires quite often because strings are a form of user interface. We tend to change them slightly and embellish them over time. For example, do we care that there is a "." At the end of the string? Out test requires us to care, but the "meat" of the assert really is about the correct number being shown (especially since strings change in different computer languages or cultures, but numbers usually stay the same).

The second test simply looks for the "2 failed" string inside the message. This makes the test more future proof - the string might change slightly but the core message remain without forcing us to change the test.

8.5 Summary

Tests grow and change with the system under tests. If we don't pay attention to maintainability, the tests may require so many changes from us that it might not be worth it to change them, and we'll end up deleting them, and throwing away all the hard work that went to create them.

To make sure they are useful in the long run, we make sure we only care about the tests when they fail for reasons we truly care about.

9 Readability

This chapter covers

- **Writing readable tests**
- **Exploring naming conventions for unit tests**

Without readability the tests you write are almost meaningless to whoever reads them later on. Readability is the connecting thread between the person who wrote the test and the poor soul who must read it a few months or years later.

Tests are stories you tell the next generation of programmers on a project. They allow a developer to see exactly what an application is made of and where it started.

This chapter is all about making sure the developers who come after you will be able to maintain the production code and the tests that you write, while understanding what they're doing and where they should be doing it.

There are several facets to readability:

- Naming unit tests
- Naming variables
- Creating good assert messages
- Separating asserts from actions

Let's go through these one by one.

9.1 Naming unit tests

Naming standards are important because they give you comfortable rules and templates that outline what you should explain about the test. No matter how I order them, or what the specific framework or language I will use, I try to make sure these three important pieces of information are present in the name of the test or in the structure of the file in which the test exists:

1. The entry point to the unit of work (or, the name of the feature you're testing)
2. The scenario under which you're testing the entry point
3. The expected behavior of the exit point

- The name of the entry point for the test (or unit of work)—

  This is essential, so that you can easily understand what's the starting scope of the logic being tested. Having this as the first part of the test name also allows easy navigation and as-you-type completion (if your IDE supports it) in the test file.
- The scenario under which it's being tested—

  This part gives you the "with" part of the name: "When I call this entry point X with a null value, then it should do Y."
- The expected behavior from the exit point of the test when the scenario is invoked—
- This part specifies in plain English what the unit of work should do or return, or how it should behave, based on the current scenario: "When I call entry point X with a null value, then it should do Y as visible from this exit point"

These parts have to exist *somewhere* close to the eyes of the person reading the test. Sometimes they can be all encapsulated in the function name of the test, sometimes you can nest them with nested *describe* structures. Sometimes you can simply use a string description as a parameter for the test name.

Here are some examples, all with the same pieces of information, but laid out differently.

Listing 9.1: Same information, different variations

```
#1
test('verifyPassword, with a failing rule, returns error based on rule.reason', () => {…}

#2
describe(verifyPassword, () => {
  describe('with a failing rule', () => {
    it(returns error based on the rule.reason', () => {…}

#3
   verifyPassword_withFailingRule_returnsErrorBasedonRuleReason()
```

You can of course come up with other ways to structure this (who says you have to use underscores? That's just my own preference to remind me and others that there are three pieces of information) – but the key point to take away is that if you remove one of these pieces of information, you're forcing the person reading the test to read the code inside the test to find out the answer, wasting precious time.

Listing 9.2: Names with missing infomation

```
Examples:
test(failing rule, returns error based on rule.reason', () => {…}
"Excuse me, what is the thing under test?"

test('verifyPassword, returns error based on rule.reason', () => {…}
"Sorry, when is this supposed to happen?"
```

```
test('verifyPassword, with a failing rule', () => {…}
“Um, what’s supposed to happen then?”
```

Your main goal with readability is to release the next developer from the burden of reading the test code in order to understand what the test is testing.

Another great reason to include all these pieces of information in the name of the test somewhere, is that the name is usually the only thing that shows up when an automated build pipeline fails. You’ll see all the names of the failed tests in the log of the build that filed, but you won’t see any comments or the code of the tests. If the names are good enough you might not need to read the code of the tests or debug them, simply by reading the log of the failed build. It can again save precious debugging and reading time.

A good test name also serves to contribute to the idea of executable documentation – if you can ask a developer who is new to the team to read the tests so they can understand how a specific component or application works, it’s a good sign of readability. If they can’t make sense of the applicational or component’s behavior from the tests alone it might be a red flag for readability.

9.2 Magic Values and Naming variables

Have you heard the term "Magic values"? It sounds awesome, but it's the opposite of that. It should really be “voodoo values” - to denote the negative effects of using these. What are they, you ask? Let’s take this test as an example to start with:

Listing 9.3: A test with magic values

```
1    describe('password verifier', () => {
2      test('on weekends, throws exceptions', () => {
3        expect(() => verifyPassword('jhGGu78!', [], 0))
4          .toThrowError("It's the weekend!");
5      });
6    });
```

Line 3 contains three magic values. Can a reader (a person that didn't write the test and doesn't know the API being tested) easily understand what the ‘0’ value in line 3 means? How about the ‘[]’ array? The first parameter to that function kind of looks like a password but even that has a magical quality to it. Let’s discuss:

- `‘0’` → This could mean so many things. As the reader I might have to search around in the code, or jump into the signature of the called function to understand that this means he day of the week.
- `‘[]’` →Again forces me to look at the signature of the called function to understand it means the password verification rules array (no rules)
- `'jhGGu78!’` → seems to be an obvious password value, but the big question I’ll have as a reader is “why this specific value? What’s important about this specific password? It’s obviously important to use this value and not any other for this test, because it seems so damned specific. (It isn’t, but the reader doesn’t know this. They’ll likely

end up using this password in other tests just to be safe. Magic values tend to propagate in the tests.)

Here’s the same test with the magic values fixed:

Listing 9.4: Fixing magic values

```
ch3-stubs/stub-time/00-parameters/password-verifier-time00.spec.js

describe("verifier2 - dummy object", () => {
  test("on weekends, throws exceptions", () => {
    const SUNDAY = 0, NO_RULES = [];
    expect(() => verifyPassword2("anything",
                                 NO_RULES,
                                 SUNDAY))
                    .toThrowError("It's the weekend!");
  });
});
```

By putting magic values into meaningfully named variable names we can remove the questions people will have when reading our test. For the password value I’ve decided to simply change the direct value to explain to the reader *what is not important* about this test.

Variable names and values are just as much about explaining to the reader *what they should NOT care about* as it is about explaining what IS important.

9.3 Separating asserts from actions

This is a short section but an important one nonetheless. For the sake of readability, and all that is holy, avoid writing the assert\expect line and the method call in the same statement.

Here’s what I mean:

Listing 9.5: separating assert from action

```
 // #Example 1
expect(verifier.verify("any value")[0]).toContain("fake reason");


 // #Example 2
const result = verifier.verify("any value");

expect(result[0]).toContain("fake reason");
```

See the difference between the two examples? Example 1 is much harder to read and understand in the context of a real test, because of the length of the line, and the nesting of the “act” part and the assert part.

Plus, it’s much easier to debug the 2nd example than it is the first one, if you wanted to focus on what is the result value after the call. Don’t skimp on this small tip. The people after you will whisper a small thank you when your test doesn’t make them feel stupid for not understanding it.

9.4 Setting up and tearing down

Setup and teardown methods in unit tests can be abused to the point where the tests or the setup and teardown methods are unreadable. Usually the situation is worse in the setup method than the teardown method.

Let's look at one possible abuse which is very common – using the setup (or "beforeEach" function) for setting up mocks or stubs.

Listing 9.6: Using a setup/beforeEach function for mock setup

```
01    describe("password verifier", () => {
02      let mockLog;
03      beforeEach(() => {
04        mockLog = Substitute.for<IComplicatedLogger>();
05      });
06
07      test("verify, with logger & passing,calls logger with PASS",() => {
08        const verifier = new PasswordVerifier2([], mockLog);
09        verifier.verify("anything");
10
11        mockLog.received().info(
12          Arg.is((x) => x.includes("PASSED")),
13          "verify"
14        );
15      });
16    });
17
```

If you have mocks and stubs being set up in a setup method, that means they don't get set up in the actual test. That, in turn, means that whoever is reading your test may not even realize that there are mock objects in use or what the expectations from them are in the test.

In lines 8 and 11 the `mockLog` is used but is initialized in the `beforeEach` function (a setup function). Imagine you had dozens or more of these tests in the file. The setup function is way up in the beginning of the file, and you are stuck reading a test way down in the file. Now you come across the `mockLog` variable and you have to start asking questions such as "where is this initialized? What will it behave line in the test?" and more.

Another problem that can arise if you have multiple mocks and stubs used in various tests in the same file, is that file initializing all of them in the setup function, the setup function becomes a dumping group for putting in all the various state used by your tests. It becomes a big mess, a soup of many parameters, some used by some tests and some used by other tests. It becomes technical debt to manage it and understand it.

It's much more readable to initialize mock objects directly in the test itself, with all their expectations. Here's an example of using the mock in each test.

Listing 9.7: avoiding a setup function

```
describe("password verifier", () => {
  test("verify, with logger & passing,calls logger with PASS",() => {
    const mockLog = Substitute.for<IComplicatedLogger>();

    const verifier = new PasswordVerifier2([], mockLog);
    verifier.verify("anything");

    mockLog.received().info(
      Arg.is((x) => x.includes("PASSED")),
      "verify"
    );
  });
```

Now, as I look at the test, everything is clear as day – I can see when the mock is created, its behavior and anything else I need to know. If you're worried about maintinability, you can refactor the creation of the mock into a helper function, which each test calls. That way, you're avoiding the generic setup function, and instead calling the same helped function form multiple tests. You keep the readability and gain more maintainability.

Listing 9.8: using a helper function

```
describe("password verifier", () => {
  test("verify, with logger & passing,calls logger with PASS",() => {
    const mockLog = makeMockLogger(); //defined at top of file

    const verifier = new PasswordVerifier2([], mockLog);
    verifier.verify("anything");

    mockLog.received().info(
      Arg..((x) => x.includes("PASSED")),
      "verify"
    );
  });
```

And yes, if you follow this logic you can see that I'm perfectly OK with you not having *any* setup functions in your tests files at all most of the time. I've many times written full test suites that didn't have a setup function, only helper methods being called from each test, for the sake of maintainability. The tests were still readable and maintainable.

9.5 Summary

In the end, it's simple: readability goes hand in hand with maintainability and trustworthiness. People who can read your tests can understand them and maintain them, and they'll also trust the tests when they pass. When this point is achieved, you're ready to handle change and to change the code when it needs changing, because you'll know when things break.

In the next chapters, we'll take a broader look at unit tests as part of a larger system: how to fit them into the organization and how they fit in with existing systems and legacy code.