

Author Picks

FREE



# Exploring Productivity Tools

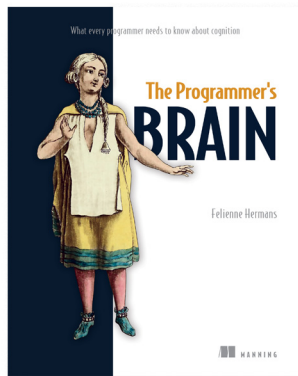
Chapters selected by Felienne Hermans



**Save 50%** on these books  
—eBook, pBook, and MEAP.

Enter **meept50** in the Promotional Code box  
when you checkout.

Only at [manning.com](http://manning.com).

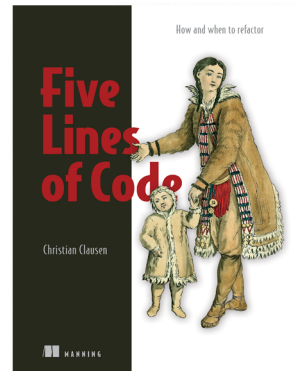


*The Programmer's Brain*

by Felienne Hermans

ISBN 9781617298677

275 pages / \$39.99

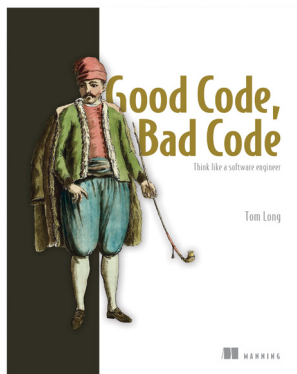


*Five Lines of Code*

by Christian Clausen

ISBN 9781617298318

275 pages / \$39.99

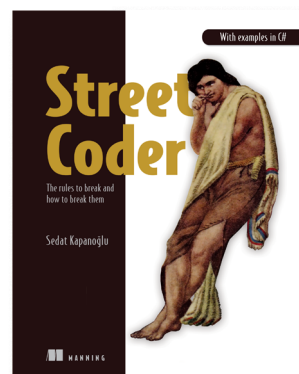


*Good Code, Bad Code*

by Tom Long

ISBN 9781617298936

350 pages / \$39.99



*Street Coder*

by Sedat Kapanoglu

ISBN 9781617298370

325 pages / \$39.99

Licensed to Sathish Jayapal <sathishk.dot@gmail.com>



## *Exploring Productivity Tools*

Chapters chosen by Felienne Hermans

Copyright 2021 Manning Publications

To pre-order or learn more about these books go to [www.manning.com](http://www.manning.com)

For online information and ordering of these and other Manning books, please visit [www.manning.com](http://www.manning.com). The publisher offers discounts on these books when ordered in quantity.

For more information, please contact


Special Sales Department  
Manning Publications Co.  
20 Baldwin Road  
PO Box 761  
Shelter Island, NY 11964  
Corporate Program Manager: Candace Gillhoolley, [corp-sales@manning.com](mailto:corp-sales@manning.com)

©2021 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.  
20 Baldwin Road Technical  
PO Box 761  
Shelter Island, NY 11964

Cover designer: Matt Bailey

ISBN: 9781633439900

# *contents*

---

*welcome*   *iv*

**How to read complex code**   **2**

Chapter 4 from *The Programmer's Brain*

**Unit testing practices**   **20**

Chapter 11 from *Good Code, Bad Code*

**Work with the compiler**   **58**

Chapter 7 from *Five Lines of Code*

**Tasty testing**   **80**

Chapter 4 from *Street Coder*

*index*   *108*

# welcome

---

At the end of the day, we all want to have accomplished something worthwhile and spent our time as productively as possible. Of course, being productive takes conscious effort, deliberate intentions, and a healthy measure of stick-to-itiveness. But despite our best-laid plans, sometimes even all those things aren't enough to move the needle as far forward on our goals as we'd hoped. That's where great productivity tools come in.

For this sampler, I've chosen chapters from four Manning books that spotlight productivity-boosting tools and techniques. The first is from my own book, *The Programmer's Brain*, and explores memory-supporting tools that will make reading complex code easier and faster. Next up is a chapter from Tom Long's *Good Code, Bad Code*, which outlines time-saving (and headache-saving) unit testing practices that help prevent errors that can happen when changes are made—and if there's one thing we as programmers can count on, it's that change is constant.

In a chapter from *Five Lines of Code* by Christian Clausen, learn how to get along with the compiler, an indispensable tool that, despite its occasional (or not so occasional) nagging, helps you get the job done more effectively and with fewer errors. Lastly, the “Tasty Testing” chapter from *Street Coder* by Sedat Kapanoglu delves into why we hate testing, how we can learn to love it, and leveraging it to decrease our workload and—you guessed it!—increase our productivity.

I hope the tools featured in this sampler help you to work smarter and not harder, and that by the end, you've experienced the value of tools like these for increasing not only your productivity but your job satisfaction and your work/life balance. If you're interested in finding out more about the tools and topics introduced here, I highly recommend the complete versions of the books they were sampled from. Thanks for reading, and I hope it will make you more productive so you can enjoy life and work!

— Felienne Hermans

Chapter 4 from *The Programmer's Brain*  
by Feliene Hermans

**I**n this chapter, you'll discover a tool that supports your working memory, cutting the time it takes to read complex, calculation-heavy code.

# *How to read complex code*

---

## ***This chapter covers***

- Analyzing when working memory is overloaded by very complex code
- Comparing and contrasting two different types of working memory overload
- Refactoring code for readability to compensate for an overloaded working memory
- Creating a state table and dependency graph to support your working memory

Chapter 1 introduced the different ways in which code can be confusing. We've seen that confusion can be caused by a lack of information, which must be acquired and stored in your short-term memory, or by a lack of knowledge, which requires storing information in your long-term memory. This chapter covers the third source of confusion: a lack of processing power in the brain.

Sometimes the code you are reading is just too complex for you to fully understand. Because reading code is not an activity that most programmers practice often, you might find yourself lacking strategies to deal with reading code that you do not understand. Common techniques include “read it again” and “give up,” neither of which is that helpful.



This chapter dives into the cognitive processes that underlie the “processing power” of the brain, which we commonly call the *working memory*. We will explore what working memory is, and how code can be so confusing that it overloads your working memory. After we’ve covered the basics, I’ll show you three techniques to support your working memory so you can process complex code with more ease.

In the previous chapters, we covered techniques to help you read code better. In chapter 2 you learned about techniques for more effective chunking of code, and chapter 3 provided tips for storing more syntax knowledge in your long-term memory, which also aids in reading code. However, sometimes code is so complex that even with a lot of syntax knowledge and efficient chunking strategies, it’s still too hard to process.

## 4.1 Reading complex code

In chapter 1, I showed you an example of a BASIC program whose execution was complicated enough that you probably couldn’t process it all just by reading the code. In such a case, you might be tempted to scribble intermediate values next to the code, as shown in figure 4.1.

```

1 LET N2 = ABS (INT (N))
2 LET B$ = ""
3 FOR N1 = N2 TO 0 STEP 0
4     LET N2 = INT (N1 / 2)
5     LET B$ = STR$ (N1 - N2 * 2) + "0"
6     LET N1 = N2
7 NEXT N1
8 PRINT B$
9 RETURN

```

**Figure 4.1** A program converting the number *N* into a binary representation in BASIC. The program is confusing because you cannot see all the small steps that are being executed. If you need to understand all the steps, you may use a memory aid like writing down the intermediate values of the variables.

The fact that you feel the need to do this means that your brain lacks the capacity to process the code. Let’s compare the BASIC code to the second example from chapter 1, a Java program that calculates the binary representation of an integer *n*. While interpreting this code might also take some mental energy, and it’s possible that unfamiliarity with the inner workings of the `toBinaryString()` method may cause you some confusion, it is unlikely that you’ll feel the need to make notes while reading it.

### Listing 4.1 Java program to convert *n* to a binary representation.

```

toBinaryString().
public class BinaryCalculator {
    public static void main (Int n) {
        System.out.println(Integer.toBinaryString(n));
    }
}

```

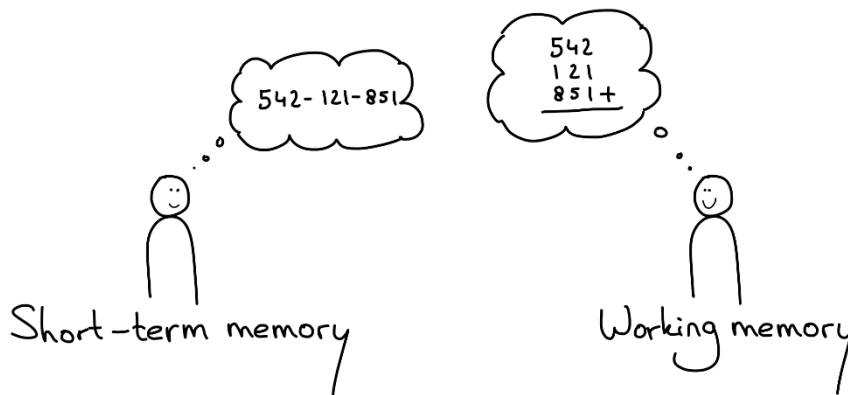
In the previous chapters we delved into two of the cognitive processes at play when you read complex code: short-term memory, which stores information briefly, and long-term memory, which stores knowledge for a longer period of time. To understand why you sometimes need to offload information, you need to understand the third cognitive process introduced in chapter 1, which we have not yet discussed in detail. The *working memory* represents the brain's capacity to think, to form new ideas, and to solve problems. Earlier, we compared the short-term memory to the RAM of a computer and the long-term memory to the hard drive. Following that analogy, the working memory is like the brain's processor.

#### 4.1.1 What's the difference between working memory and short-term memory?

Some people use “working memory” as a synonym for “short-term memory,” and you might have seen the two terms used interchangeably. Others, however, distinguish between the two concepts, and we will do that in this book. The role of the short-term memory is to *remember* information. The role of the working memory, on the other hand, is to *process* information. We will treat these two processes as separate.

**DEFINITION** The definition of working memory that we will use in the remainder of this book is *the short-term memory applied to a problem*.

Figure 4.2 shows an example of the difference between the two processes: if you are remembering a phone number you use your short-term memory, whereas if you are adding integers, you use your working memory.



**Figure 4.2** The short-term memory briefly stores information (like a phone number, as shown on the left), while the working memory processes information (like when performing a calculation, as shown on the right).

As you saw in chapter 2, the short-term memory can typically only hold 2 to 6 items at a time. More information can be processed when the information is divided into recognizable chunks, like words, chess openings, or design patterns. Because the working memory is the short-term memory *applied to a certain problem*, it has the same limitation.

## 4.2 What happens in the working memory when you read code?

Like the short-term memory, the working memory is only capable of processing between 2 and 6 things at a time. In the context of working memory, this capacity is known as the *cognitive load*. When you are trying to solve a problem that involves too many elements that cannot be divided efficiently into chunks, your working memory will become “overloaded.”

This chapter will introduce methods to systematically address cognitive load, but before we can cover these techniques, we need to explore the different types that exist. The researcher who first proposed cognitive load theory was the Australian professor John Sweller. Sweller distinguished three different types of cognitive load: intrinsic, extraneous, and germane. Table 4.1 shows a quick summary of how cognitive load varies.

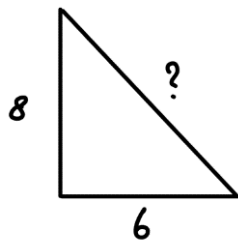
**Table 4.1** Types of Cognitive Load

Load type	Brief explanations
Intrinsic load	How complex the problem is in itself
Extraneous load	What outside distractions add to the problem
Germane load	Cognitive load created by having to store your thought to long-term memory

We will focus on the first two types of cognitive load here; we discuss germane load more in-depth in a later chapter.

### 4.2.1 Intrinsic cognitive load when reading code

*Intrinsic cognitive load* is cognitive load caused by features of a problem that the problem contains by nature. For example, imagine that you have to calculate the hypotenuse of a triangle, as illustrated by figure 4.3.



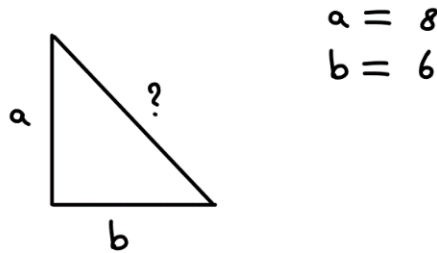
**Figure 4.3** A geometry problem in which the lengths of two sides of a triangle are given and the third one needs to be calculated. This can be a problem that is hard to solve, depending on your prior knowledge. However, the problem itself cannot be made any simpler without changing it.

Solving this calculation has certain characteristics that are inherent to the problem. For example, you need to know Pythagoras’s theorem ( $a^2 + b^2 = c^2$ ) to solve it, and you have to be able to calculate first the squares of 8 and 6 and then the square root of the sum of the results. Because there is no other way to solve the problem or to simplify these steps, this load is *intrinsic* to the problem. In programming, we often use the term *inherent complexity* to describe these intrinsic aspects of a problem. In cognitive science, we say that these aspects cause cognitive load of the intrinsic type.

### 4.2.2 Extraneous cognitive load

In addition to the natural, intrinsic load that problems can cause in the brain, there is also cognitive load that is *added* to problems, often by accident. For example, in figure 4.4 the same question about finding the length of the hypotenuse is formulated in a different way, requiring us to make a mental connection between the labels for the two sides of the triangle whose lengths are known and their values. This additional work results in a higher *extraneous cognitive load*.

Consider these values for a and b:



**Figure 4.4** This way of formulating the problem of finding the length of the third side of a triangle incurs a higher extraneous cognitive load.

Solving the problem has not really been made harder—we still need to remember and apply Pythagoras’s theorem. However, our brains do need to work harder on extraneous tasks: connecting *a* to the value 8 and *b* to the value 6. In programming, we can think of this type of extraneous load as similar to accidental complexity: aspects of a program that make a problem harder than it needs to be.

#### EXTRANEOUS LOAD AND PROGRAMMING

What creates extraneous load is not the same for every programmer. The more experience you have using a certain concept, the less cognitive load it creates for you. For example, the two Python code examples in the following listing are computationally equivalent.

**Listing 4.2 Two versions of a Python program to select all items above 10, one with and one without list comprehensions**

```
above_ten = [a for a in items if a > 10]

above_ten = []
for a in items:
    if a > 10: new_items.append(a)
```

Because the two code snippets both solve the same problem, they share the same intrinsic cognitive load. However, whether they represent the same extraneous cognitive load for you depends on your prior knowledge: if you’re not familiar with list comprehensions, the extraneous load caused by the first example will be much higher than for someone who is experienced using them.

**EXERCISE 4.1**

The next time you read unfamiliar code, try to monitor your own cognitive load. When the code is hard to process and you feel the need to make notes or follow the execution step by step, it is likely that you are experiencing a high cognitive load.

When you are in a situation where you are experiencing high cognitive load, it is worthwhile to examine which parts of the code are creating the different types of cognitive load. You can use the following table to analyze this.

Lines of code	Intrinsic cognitive load	Extraneous cognitive load

**4.3 Refactoring code to reduce cognitive load**

Now that you know the different ways code can overload the working memory, it's time to direct our attention to ways of lowering cognitive load. The remainder of this chapter discusses three methods that will make it easier for you to read complex code. The first technique is one that you might already be familiar with, albeit in a different context: *refactoring*.

A refactoring is a transformation of code that changes its internal structure without changing the code's external behavior. For example, if a certain block of code is very long you might want to split the logic into multiple functions, or if the codebase exhibits duplication you might want to refactor it to gather all the duplicated code in one place for easy reuse. In the following Python code listing, for example, the repeated calculation could be placed into a method.

**Listing 4.3 Python code that repeats the same calculation twice**

```
vat_1 = Vat(waterlevel = 10, radius = 1)
volume_vat_1 = math.pi * vat_1.radius **2 * vat_1. water_level
print(volume_vat_1)

vat_1 = Vat(waterlevel = 25, radius = 3)
volume_vat_2 = math.pi * vat_2. radius **2 * vat_2. water_level
print(volume_vat_2)
```

In most cases, refactoring is done to make it easier to maintain the resulting code. For example, eliminating duplicate code means changes to that code only have to be made in one location.

But code that is more maintainable in the long run is not always more readable now. For example, consider a piece of code that contains many method calls, and thus depends on code spread out over many different places in a file, or even in multiple files. That architecture might be more maintainable because all the logic has its own method. However, such *delocalized* code can also be harder on your working memory, because you will have to scroll or search for function definitions in different locations.

Therefore, sometimes you might want to refactor code not to make it more maintainable in the long run, but to make the code more readable for you at this point in time. That process can sometimes involve “reverse refactoring”, which decreases maintainability, such as inlining a method (as discussed in the following subsection). In many cases these refactorings are temporary, only meant to allow you understand the code and can be rolled back once your understanding is solidified.

While this process might have been a big hassle a few years ago, version control systems are now used for most codebases and are integrated into most IDEs, which makes it relatively easy to start a local “understanding” branch where you execute the changes needed to comprehend the code. And if some of your refactorings turn out to be valuable in a broader sense, they can be merged in with relative ease.

#### **4.3.1 Inlining code**

An example of a refactoring that can be used as an aid in comprehending code is the inlining of a method or function. This can be especially helpful when a method’s name is not very revealing, such as `calculate()` or `transform()`.

When reading a call to a method with a vague name, you will need to spend some time up front to understand what the method does—and it will likely take several exposures before the functionality of the method is stored in your long-term memory. Inlining the method lowers your extraneous cognitive load and might help you to comprehend the code that calls the method. Additionally, studying the code of the method itself might help you to understand the code, which can be easier with more context. Within the new context, you might also be able to choose better name for the method.

Alternatively, you might want to reorder methods within the code—for example, code may be easier to read if a method’s definition appears close to the first method call. Of course, many IDEs nowadays have shortcuts to navigate to method and function definitions, but using such a function also takes up a bit of working memory and might cause additional extraneous cognitive load.

#### **4.3.2 Replacing unfamiliar language constructs**

The remainder of this chapter covers techniques that can help you combat the three possible sources of confusion when reading code (lack of knowledge, information, and processing power). If the code you are reading contains programming concepts that you are not familiar with, the problem you are dealing with is a lack of knowledge. We’ll start with a technique that can be helpful in these cases.

In some situations, the unfamiliar constructs you are working with could be expressed in a different, more familiar way. For example, many modern programming languages

(like Java and C#) support *anonymous functions*, often referred to as *lambdas*. Lambdas are functions that do not need to be given a name (hence “anonymous”). Another example is a list comprehension in Python. Lambdas and list comprehensions are great ways to make code shorter and more readable, but many programmers are not familiar with them and read them with less ease than they would read a `for` or `while` loop.

If the code you are reading or writing is simple and straightforward, lambdas or list comprehensions might not pose a problem—but if you are working with more complex code such as advanced structures might cause your working memory to become overloaded. Less familiar language constructs increase the extraneous cognitive load on your working memory, so when you are reading complex code it can be beneficial for your working memory to not have to deal with these.

While the precise language constructs that you might want to replace are, of course, dependent on your own prior knowledge, there typically are two reasons to replace code to lower your cognitive load: first because these constructs are known to be confusing, and second because they have a clear equivalent that is more basic. Both conditions apply to lambdas and list comprehensions, so these are good examples to use to demonstrate this technique. It can be useful to translate these to a `for` or `while` loop to lower the cognitive load until you gain more understanding of what the code is doing. Ternary operators are also a good candidate for such a refactoring.

#### LAMBIDAS

The Java code in the following listing is an example of an anonymous function used as a parameter for a `filter()` function. If you are familiar with the use of lambdas, this code will be easy enough to follow.

#### Listing 4.4 A `filter()` function in Java that takes an anonymous function as an argument

```
Optional<Product> product = productList.stream().
    filter(p -> p.getId() == id).
    findFirst();
```

However, if lambdas are new to you, this code might cause too much extraneous cognitive load. If you feel that you are struggling with the lambda expression, you can simply rewrite the code to use a regular function temporarily, as in the following example.

#### Listing 4.5 `filter()` function in Java that uses a traditional function as an argument

```
public static class Toetsie implements Predicate <Product>{
    private int id;

    Toetsie(int id){
        this.id = id
    }

    boolean test(Product p){
        return p.getID() == this.id;
    }
}
```

```
Optional<Product> product = productList.stream().
    filter(new Toetsie(id)).
    findFirst();
```

### LIST COMPREHENSIONS

Python supports a syntactic structure called *list comprehensions*, which can create lists based on other lists. For example, you can use the following code to create a list of first names based on a list of customers.

#### Listing 4.6 A list comprehension in Python transforming one list into another list

```
names = [c.first_name for c in customers]
```

List comprehensions can also use filters that make them a bit more complex—for example, to create a list of the names of customers over 50 years of age, as shown next.

#### Listing 4.7 A list comprehension in Python using a filter

```
names = [c.first_name for c in customers if c.age > 50]
```

While this code might be easy enough to read for someone used to list comprehensions, for someone who isn't (or even someone who is, if it's embedded in a complex piece of code) it might cause too much strain on the working memory. When that happens, you can transform the list comprehension to a for loop to ease understanding, as shown here.

#### Listing 4.8 A for loop in Python transforming one list into another list using a filter

```
names = []

for c in customers:
    if c.age > 50:
        names.append(c.first_name)
```

### TERNARY OPERATORS

Many languages support *ternary operators*, which are shorthand for if statements. They typically have the form of a condition followed by the result when the condition is true and then the result when the condition is false. For example, the following listing is a line of JavaScript code that checks whether the Boolean variable `isMember` is true using a ternary operator. If `isMember` is true, the ternary operator returns \$2.00; if not, it returns \$10.00.

#### Listing 4.9 JavaScript code using a ternary operator

```
isMember ? '$2.00' : '$10.00'
```

Some languages, such as Python, support ternary operations in a different order, giving first the result when the condition is true, followed by the condition and then the result when the condition is false. The following example is a line of Python code that checks whether the Boolean variable `isMember` is true using a ternary operator. As in the JavaScript example, if `isMember` is true the ternary operator returns \$2.00, and if it isn't it returns \$10.00.



**Listing 4.10 Python code using a ternary operator**

```
'$2.00' if is_member else '$10.00'
```

Conceptually, a ternary operator is not hard to understand; as a professional programmer you're probably familiar with conditional code. However, the fact that the operation is placed on one line or that the order of the arguments is different from in a traditional `if` statement might cause the code to create too much extraneous cognitive load for you.

**"BUT LAMBDA/ LIST COMPREHENSIONS/TERNARIES ARE MORE READABLE"**

To some people, the refactorings described in the previous sections might feel weird or wrong. You might believe that shortening code with a lambda or a ternary is always preferable because it's more readable, and you might object to the idea of refactoring code to a worse state. However, as you've seen in this chapter and earlier in this book, "readable" is really in the eye of the beholder. Just as if you're familiar with chess openings it's easy to remember them, if you're familiar with using ternaries it's easy to read code that contains them. What is easy to read depends on your prior knowledge, and there is no shame in helping yourself understand code by translating it to a more familiar form.

Depending on your codebase, however, you might want to revert the changes you've made to the code for readability purposes once you're sure you understand it. If you are new to a team and you're the only one unfamiliar with list comprehensions, you'll want to leave them in place and roll back your refactorings.

**CODE SYNONYMS ARE GREAT ADDITIONS TO A FLASHCARD DECK**

While there's no shame in changing code temporarily to aid comprehension, this does point to a limitation in your understanding. In chapter 3, you learned about creating a deck of flashcards to use as a learning and memory aid, with code on one side and a textual prompt on the other side. For example, a card on `for` loops could have "print all numbers between 0 and 10 in C++" on one side and the corresponding C++ code (shown in the following listing) on the other.

**Listing 4.11 C++ code for printing the numbers between 0 and 10**

```
for (int i = 0; i <= 10; i = i + 1) {  
    cout << i << "\n";  
}
```

If you often struggle with, for example, list comprehensions, you can consider adding a few cards on that programming construct to your deck. For more advanced programming concepts like these, it can work better to have code on both sides of the flashcard rather than a text explanation—that is, you can have the vanilla code on one side and on the other side the equivalent code using the advanced concept, such as a ternary or lambda.

### 4.3.3 **You want to consider cognitive load when writing code, too**

In this chapter we have looked at the role of cognitive load when you are reading unfamiliar or complex code. Of course, the onus of understanding code is not entirely on the reader. When writing code, you will also want to consider the cognitive load that people might experience when reading the code. That means structuring and documenting the code in a way that makes it more navigable, more chunkable, and more quickly understood by a reader. Later chapters will dive into *writing* code that is easier on the memory of readers.

## 4.4 **Marking dependencies**

The previous section introduced one technique to reduce the cognitive load that code can create: refactoring it to a more familiar form. However, even in its refactored state code might still overload your working memory if its structure is too complex.

There are two ways in which code with a complicated structure can overload the working memory. First, it might be the case that you do not know exactly which parts of the code you need to read. This causes you to read more of the code than is needed, which may be more than your working memory is able to process.

Second, with code that is highly connected, your brain is trying to do two things at the same time: understand individual lines of code and understand the structure of the code to decide where to continue reading. For example, when you encounter a method call of which you do not know the exact functionality, you might need to locate and read the method before you can continue reading the code at the call site.

If you have ever read the same piece of code five times in a row without making progress, you were probably in a situation where you didn't know what parts of the code to focus on, and in what order. You may have been able to understand each line of code individually but lacked an understanding of the bigger picture.

When you reach the limits of your working memory, you can use a memory aid to help you focus on the right parts of the code. Creating a dependency graph on top of your code can help you to understand the flow and help you to read the code by following the logical flow.

For this technique, I would advise you to print out the code, or convert it to a PDF and open it on a tablet so you can make annotations digitally. Follow these steps to annotate the code to support your memory in processing it:

- 1 Circle all the variables.

Once you have the code in a form you can annotate, start by finding all the variables and circling them, as shown in figure 4.5.

- 2 Link similar variables.

Once you have located all the variables, draw lines between occurrences of the same variable, as illustrated in figure 4.6. This helps you to understand where data is used in the program. Depending on the code, you may also want to link similar variables (for example, accesses into a list, as in `customers[0]` and `customers[i]`).

```

digits = "0123456789abcdefghijklmnopqrstuvwxyz"

def baseN(num,b):
    if num == 0: return "0"
    result = ""
    while num != 0:
        num, d = divmod(num, b)
        result += digits[d]
    return result[::-1] # reverse

def pal2(num):
    if num == 0 or num == 1: return True
    based = bin(num)[2:]
    return based == based[::-1]

def pal_23():
    yield 0
    yield 1
    n = 1
    while True:
        n += 1
        b = baseN(n, 3)
        revb = b[::-1]
        #if len(b) > 12: break
        for trial in ('{0}{1}'.format(b, revb), '{0}0{1}'.format(b, revb),
                      '{0}1{1}'.format(b, revb), '{0}2{1}'.format(b, revb)):
            t = int(trial, 3)
            if pal2(t):
                yield t

```

Figure 4.5 Code in which all variables are circled to support understanding.

```

digits = "0123456789abcdefghijklmnopqrstuvwxyz"

def baseN(num,b):
    if num == 0: return "0"
    result = ""
    while num != 0:
        num, d = divmod(num, b)
        result += digits[d]
    return result[::-1] # reverse

def pal2(num):
    if num == 0 or num == 1: return True
    based = bin(num)[2:]
    return based == based[::-1]

def pal_23():
    yield 0
    yield 1
    n = 1
    while True:
        n += 1
        b = baseN(n, 3)
        revb = b[::-1]
        #if len(b) > 12: break
        for trial in ('{0}{1}'.format(b, revb), '{0}0{1}'.format(b, revb),
                      '{0}1{1}'.format(b, revb), '{0}2{1}'.format(b, revb)):
            t = int(trial, 3)
            if pal2(t):
                yield t

```

Figure 4.6 Code in which all variables are circled and linked to their other occurrences to support understanding.

Linking all the variables will help you read the code, because instead of searching for other occurrences you can simply follow the lines. This lowers your cognitive load and thus frees up working memory for you to focus on the functionality of the code.

**3** Circle all method/function calls.

Once you have located all the variables, focus on the methods and functions in the code. Circle them in a different color.

**4** Link methods/functions to their definitions.

Draw a line between each function or method definition and the locations where they are invoked. Focus special attention on methods with just one invocation, because these methods are candidates to be inlined with a refactoring as explained earlier in this chapter.

**5** Circle all instances of classes.

Once you have located the variables and functions, focus on classes. Circle all instances of classes in a third color.

**6** Draw a link between classes and their instances.

As a final step in examining the code, link instances of the same class to their definition, if that definition is present in the code. If the definition is not present, you can link the instances of the same class to each other.

The colored pattern you have created using the previous six steps indicates the *flow* of the code and can be used as an aid in reading it. You now have a reference that you can refer to for information about the code's structure, saving you the effort of, for example, having to search for definitions while also deciphering the meaning of the code, which can overload your working memory. You can start at an entry point of the code, such as the `main()` method, and read it from there. Whenever you encounter a link to a method call or class instantiation, you can follow the line you drew and continue reading at the right place straight away, avoiding wasting time searching or reading more code than needed.

## **4.5 Using a state table**

Even when code is refactored to the easiest possible form aligned with your prior knowledge, and with all dependencies marked, it can still be confusing. Sometimes the cause of the confusion is not the structure of the code, but the calculations it performs. The issue here is a lack of processing power.

For example, let's revisit again the BASIC code from chapter 1 that converts the number `N` into a binary representation. In this program the variables influence each other heavily, so a detailed examination of their values is required to understand it. You can use a memory aid like a dependency graph for code like this that performs complicated calculations, but there's another tool that can help with calculation-heavy code: a *state table*.

A state table focuses on the values of variables rather than the structure of the code. It has columns for each variable and lines for each step in the code. Take another look at our example BASIC program in listing 4.13. It's confusing because you cannot see all the intermediate calculations and their effects on the different variables.

**Listing 4.12 BASIC code that converts a number N to its binary representation**

```

1 LET N2 = ABS (INT (N))
2 LET B$ = ""
3 FOR N1 = N2 TO 0 STEP 0
4     LET N2 = INT (N1 / 2)
5     LET B$ = STR$ (N1 - N2 * 2) + B$
6     LET N1 = N2
7 NEXT N1
8 PRINT B$
9 RETURN

```

If you need to understand code like this with many interconnected calculations, you can use a memory aid like the partial state table shown in figure 4.7.

	N	N2	B\$	N1
init	7	7	-	7
loop1		3	0	3
loop2				

**Figure 4.7** An example of a partial state table for the BASIC code for calculating the binary representation of a number.

Follow these steps to create a state table:

- 1 Make a list of all the variables.

If you have already created a dependency graph for this program, as described in the previous section, it will be easy to list the variables because you'll have circled all of them in the same color.

- 2 Create a table and give each variable its own column.

In the state table, each variable will get one column in which its intermediate values can be recorded, as shown in figure 4.7.

- 3 Add one row to the table for each distinct part of the execution of the code.

Code that contains complex calculations will most likely also contain some complex dependencies, such as a loop depending on a calculation, or a complicated if statement. Rows in the state table represent separate parts of the dependencies. For example, as shown in figure 4.7, a row can represent one iteration in a loop, preceded by the initialization code. Alternatively, a row could represent a branch in a large if statement, or simply a group of coherent lines of code. In extremely complex and terse code, one row in the table might even represent one line of code.

- 4 Execute each part of the code and write down the value each variable has afterward in the correct row and column.

Once you've prepared the table, work your way through the code and calculate the new value of each variable for each row in the state table. The process of mentally executing code is called *tracing* or *cognitive compiling*. When tracing the code using a state table, it can be tempting to skip a few variables and only fill in part of the table—but try to resist that temptation. Working through it meticulously will help you gain a deeper understanding of the code, and the resulting table will support your overloaded working memory. On a second read of the program you can use the table as a reference, allowing you to concentrate on the coherence of the program rather than on the detailed calculations.

### An app to support the working memory

Manual creation of visualizations to support your working memory has a lot of value because it forces you to examine the code in detail. However, these visualizations can also be created automatically. An elegant program for this purpose is Python Tutor, created by Philip Guo, professor of cognitive science at the University of California, San Diego. Python Tutor, as shown in the figure below. Python Tutor which is now available for many more programming languages than Python alone, visualizes the execution of a program. For example, the following figure shows that Python stores integers and lists differently; for an integer the value is stored, whereas for a list a pointer-like system is used.



**Figure 4.8** Python Tutor showing the difference between storing the integer `x` and its value directly, and storing a list `fruit` with a pointer.

Research exploring the use of Python Tutor in education<sup>a</sup> has shown that it takes students a while to get used to working with the program but that it is helpful, especially when debugging.

a) See “The Use of Python Tutor on Programming Laboratory Session: Student Perspectives” by Oscar Karnalim and Mewati Ayub (2017), available at <https://kinetik.umm.ac.id/index.php/kinetik/article/view/442>.

### 4.5.1 Combining state tables and dependency graphs

This section and the previous one described two techniques to support your working memory when reading code by offloading some information about the code onto paper: drawing a dependency graph and creating a state table. These techniques focus on different parts of the code. While the dependency graph draws your attention to how the code is organized, the state table captures the calculations in the code. When exploring unfamiliar code, you can use both exercises to gain a full picture of its inner workings and to use as memory aids when reading the code after completing them.

#### EXERCISE 4.2

Following the steps outlined in the previous sections, create both a dependency graph and a state table for each of the following Java programs.

##### Program 1

```
public class Calculations {
    public static void main(String[] args) {
        char[] chars = {'a', 'b', 'c', 'd'};
        // Looking for bba
        calculate(chars, 3, i -> i[0] == 1 && i[1] == 1 && i[2] == 0);
    }
    static void calculate (char[] a, int k, Predicate<int[]> decider) {
        int n = a.length;
        if (k < 1 || k > n)
            throw new IllegalArgumentException("Forbidden");

        int[] indexes = new int[n];
        int total = (int) Math.pow(n, k);

        while (total-- > 0) {
            for (int i = 0; i < n - (n - k); i++)
                System.out.print(a[indexes[i]]);
            System.out.println();

            if (decider.test(indexes))
                break;

            for (int i = 0; i < n; i++) {
                if (indexes[i] >= n - 1) {
                    indexes[i] = 0;
                } else {
                    indexes[i]++;
                    break;
                }
            }
        }
    }
}
```

##### Program 2

```
public class App {
    private static final int WIDTH = 81;
    private static final int HEIGHT = 5;
```

```

private static char[] [] lines;
static {
    lines = new char[HEIGHT][WIDTH];
    for (int i = 0; i < HEIGHT; i++) {
        for (int j = 0; j < WIDTH; j++) {
            lines[i][j] = '*';
        }
    }
}

private static void show(int start, int len, int index) {
    int seg = len / 3;
    if (seg == 0) return;
    for (int i = index; i < HEIGHT; i++) {
        for (int j = start + seg; j < start + seg * 2; j++) {
            lines[i][j] = ' ';
        }
    }
    show (start, seg, index + 1);
    show (start + seg * 2, seg, index + 1);
}

public static void main(String[] args) {
    show (0, WIDTH, 1);
    for (int i = 0; i < HEIGHT; i++) {
        for (int j = 0; j < WIDTH; j++) {
            System.out.print(lines[i][j]);
        }
        System.out.println();
    }
}
}

```

## Summary

- Cognitive load represents the limit of what the working memory can process. When you experience too much cognitive load, you cannot properly process code.
- There are two types of cognitive load that are relevant in programming. Intrinsic cognitive load is created by the inherent complexity of a piece of code, while extraneous cognitive load is added to code either accidentally (by the way it is presented) or because of gaps in the knowledge of the person reading the code.
- Refactoring is a way to reduce extraneous cognitive load, by transforming code to align better with your prior knowledge.
- Creating a dependency graph can help you understand a piece of complex and interconnected code.
- Creating a state table containing the intermediate values of variables can aid in reading code that is heavy on calculations.



Chapter 11 from *Good Code, Bad Code*  
by Tom Long

**T**his chapter explores several practical techniques for maximizing the key features of good unit tests and empowering us to confidently update and improve our code.

# 11

## *Unit testing practices*

---

### ***This chapter covers***

- Effectively and reliably unit testing all the behaviors of a piece of code
- Ensuring that tests are understandable and that failures are well-explained
- Using dependency injection to ensure that code is testable

Chapter 10 identified a number of principles that can be used to guide us toward writing effective unit tests. This chapter builds on these principles to cover a number of practical techniques that we can apply in our everyday coding.

One of the key principles in chapter 10 was the key features that good unit tests should exhibit. The motivation for many of the techniques described in this chapter directly follow from these, so as a reminder the key features are as follows:

- Accurately detects breakages—If the code is broken, a test should fail. And a test should fail only if the code is indeed broken (we don't want false alarms).
- Agnostic to implementation details—Changes in implementation details should ideally not result in changes to tests.
- Well-explained failures—If the code is broken, the test failure should provide a clear explanation of the problem.

- Understandable test code—Other engineers need to be able to understand what exactly a test is testing and how it is doing it.
- Easy and quick to run—Engineers usually need to run unit tests quite often during their everyday work. A slow or difficult to run unit test will waste a lot of engineering time.

It's by no means a given that the tests we write will exhibit these features, and it's all too easy to end up with tests that are ineffective and unmaintainable as a result. Luckily, there are practical techniques that we can apply to maximize the chance that our tests do exhibit these features. The following sections cover some of the main ones.

### **11.1 Test behaviors not just functions**

Testing a piece of code is a bit like working through a to-do list. There are things that the code under test does (or will do if we're writing the tests before writing the code) and we need to write a test case to test each of these. But as with any to-do list, a successful outcome is contingent on the correct things actually being on the list.

A mistake that engineers sometimes make is to look at a piece of code and add only function names to their to-do list of things to test. So, if a class has two functions, then an engineer might write only two test cases (one for each function). We established in chapter 10 that we should test all the important behaviors that a piece of code exhibits. The problem with concentrating on testing each function is that a function can often exhibit more than one behavior and a behavior can sometimes span across multiple functions. So, if we write only one test case per function, it's likely that we might miss some important behaviors. It's better to fill our to-do list with all the behaviors we care about, rather than just the function names we see.

#### **11.1.1 One test case per function is often inadequate**

Imagine we work for a bank maintaining a system that automatically assesses mortgage applications. The code in the following listing shows the class that makes the decision of whether a customer can get a mortgage, and if so, how much they can borrow (see listing 11.1). There are quite a few things going on in the code, such as the following:

- The `assess()` function calls a private helper function to determine whether or not the customer is eligible for a mortgage. A customer is eligible if they:
  - Have a good credit rating
  - Have no existing mortgage
  - Are not banned by the company
- If the customer is eligible, then another private helper function is called to determine the maximum loan amount for the customer. This is calculated as their yearly income minus their yearly outgoings, multiplied by 10.

**Listing 11.1 Mortgage assessment code**

```

class MortgageAssessor {
    private const Double MORTGAGE_MULTIPLIER = 10.0;

    MortgageDecision assess(Customer customer) {
        if (!isEligibleForMortgage(customer)) {
            return MortgageDecision.rejected();
        }
        return MortgageDecision.approve(getMaxLoanAmount(customer));
    }

    private static Boolean isEligibleForMortgage(Customer customer) {
        return customer.hasGoodCreditRating() &&
            !customer.hasExistingMortgage() &&
            !customer.isBanned();
    }

    private static MonetaryAmount getMaxLoanAmount(Customer customer) {
        return customer.getIncome()
            .minus(customer.getOutgoings())
            .multiplyBy(MORTGAGE_MULTIPLIER);
    }
}

```

**The application is rejected if the customer is ineligible.**

**This is a private helper function to determine if customer is eligible.**

**This is a private helper function to determine max loan amount.**

Now imagine we go and look at the tests for this code and see one single test case that just tests the `assess()` function. Listing 11.2 shows this single test case. This tests some of the things that the `assess()` function does, such as:

- A mortgage being approved for a customer with a good credit rating, no existing mortgage, and who isn't banned.
- The max loan amount being the customer's income minus their outgoings, multiplied by 10.

But it also clearly leaves a lot of things untested, such as all the reasons why a mortgage might be rejected. This is clearly an inadequate amount of testing: we could modify the `MortgageAssessor.assess()` function to approve mortgages even for banned customers and the tests would still pass!

**Listing 11.2 Mortgage assessment test**

```

testAssess() {
    Customer customer = new Customer(
        income: new MonetaryAmount(50000, Currency.USD),
        outgoings: new MonetaryAmount(20000, Currency.USD),
        hasGoodCreditRating: true,
        hasExistingMortgage: false,
        isBanned: false);
    MortgageAssessor mortgageAssessor = new MortgageAssessor();

    MortgageDecision decision = mortgageAssessor.assess(customer);
}

```

```
assertThat(decision.isApproved()).isTrue();
assertThat(decision.getMaxLoanAmount()).isEqualTo(
    new MonetaryAmount(300000, Currency.USD));
}
```

The problem here is that the engineer writing the tests has concentrated on testing functions not behaviors. The `assess()` function is the only function in the public API of the `MortgageAssessor` class, so they wrote only one single test case to test this function. Unfortunately, this one test case is nowhere near sufficient to fully ensure that the `MortgageAssessor.assess()` function behaves in the correct way.

### 11.1.2 Solution: Concentrate on testing each behavior

As the previous example demonstrates, there is often not a one-to-one mapping between functions and behaviors. If we concentrate on testing just functions, then it is very easy to end up with a set of test cases that do not verify all the important behaviors that we actually care about. In the example of the `MortgageAssessor` class there are several behaviors that we care about, including the following:

- That a mortgage application is rejected for any customers that at least one for the following applies to:
  - They don't have a good credit rating.
  - They already have an existing mortgage.
  - They are banned by the company.
- If a mortgage application is accepted, then the maximum loan amount is the customer's income minus their outgoings, multiplied by 10.

Each one of these behaviors should be tested, which requires writing a lot more than one single test case. To increase our level of confidence in the code, it also makes sense to test different values and boundary conditions, so we would probably want to include test cases such as the following:

- A few different values for incomes and outgoings to ensure that the arithmetic in the code is correct
- Some extreme values, such as zero income or outgoings, as well as very large amounts of income or outgoings

It's not unlikely that we'll end up with 10 or more different test cases to fully test the `MortgageAssessor` class. This is completely normal and expected: it's not uncommon to see 300 lines of test code for a 100-line piece of real code. It's actually sometimes a warning sign when the amount of test code doesn't exceed the amount of real code, as this can suggest that not every behavior is being tested properly.

The exercise of thinking up behaviors to test is also a great way to spot potential problems with the code. For example, as we're thinking of behaviors to test, we'll probably end up wondering what will happen if a customer's outgoings exceed their income. Currently, the `MortgageAssessor.assess()` function will approve such an

application with a negative maximum loan amount. This is kind of weird functionality, so this realization would probably prompt us to revisit the logic and handle this scenario a bit more gracefully.

#### **DOUBLE-CHECK THAT EVERY BEHAVIOR HAS BEEN TESTED**

A good way to gauge whether or not a piece of code is tested properly is to think about how someone could theoretically break the code and still have the tests pass. Some good questions to ask while looking over the code are as follows. If the answer to any of them is “yes,” then this suggests that not all the behaviors are being tested.

- Are there any lines of code that could be deleted and still result in the code compiling and the tests passing?
- Could the polarity of any if-statements (or equivalent) be reversed and still result in the tests passing? For example, swapping “if (something) {“ with “if (!something) {“
- Could any logical or arithmetic operators be replaced with alternatives and still result in the tests passing? Examples of this might be swapping a && with a || or swapping a + with a -.
- Could the values of any constants or hard-coded values be changed and still result in the tests passing?

The point is that each line of code, if-statement, logical expression, or value in the code under test should exist for a reason. If it genuinely is superfluous code, then it should be removed. If it’s not superfluous, then that means that there must be some important behavior that is somehow dependent upon it. If there is an important behavior that the code exhibits, then there should be a test case to test that behavior. So, any change in functionality to the code should result in at least one test case failing. If it doesn’t, then not all the behaviors are being tested.

The only real exception to this is code that defensively checks for programming errors. For example, we might have a check or assertion within the code to ensure that a particular assumption is valid. There may be no way to exercise this in a test, because the only way to test the defensive logic would be to break the assumption by breaking the code.

Checking that changes in functionality result in a test failure can sometimes be automated to some extent using *mutation testing*. A mutation testing tool will create versions of the code with small things mutated. If the tests still pass after the code has been mutated, then this is a sign that not every behavior is tested properly.

#### **DON’T FORGET ABOUT ERROR SCENARIOS**

Another important set of behaviors that it can be easy to overlook is how the code behaves when error scenarios occur. These can seem a bit like edge cases, because we don’t necessarily expect errors to occur that often. But how a piece of code handles and signals different error scenarios are nonetheless important behaviors that we (and callers of our code) care about. They should therefore be tested.

To demonstrate this, consider the following code listing. The `BankAccount.debit()` function throws an `ArgumentException` if it's called with a negative amount. The function being called with a negative amount is an error scenario, and the fact that it throws an `ArgumentException` when this happens is an important behavior. It should therefore be tested.

### Listing 11.3 Code that handles an error

```
class BankAccount {
    ...
    void debit(MonetaryAmount amount) {
        if (amount.isNegative()) {
            throw new ArgumentException("Amount can't be negative");
        }
        ...
    }
}
```

**Throws an `ArgumentException` is the amount is negative.**

The following listing shows how we might test the behavior of the function in this error scenario. The test case asserts that an `ArgumentException` is thrown when `debit()` is called with an amount of `-$0.01`. It also asserts that the thrown exception contains the expected error message.

### Listing 11.4 Testing error handling

```
void testDebit_negativeAmount_throwsArgumentException {
    MonetaryAmount negativeAmount =
        new MonetaryAmount(-0.01, Currency.USD);
    BankAccount bankAccount = new BankAccount();

    Exception exception = assertThrows(
        ArgumentException,
        () -> bankAccount.debit(negativeAmount))
    assertThat(exception.getMessage())
        .isEqualTo("Amount can't be negative");
}
```

**Asserts that an `ArgumentException` is thrown when `debit()` is called with a negative amount.**

**Asserts that the thrown exception contains the expected error message.**

A piece of code tends to exhibit many behaviors, and it's quite often the case that even a single function can exhibit many different behaviors depending on the values it's called with or the state that the system is in. Writing just one test case per function rarely results in an adequate amount of testing. Instead of concentrating on functions, it's usually more effective to identify all the behaviors that ultimately matter and ensure that there is a test case for each of them.

## 11.2 Avoid making things visible just for testing

A class (or unit of code) usually has some number of functions that are visible to code outside, we often refer to these as being *public* functions. This set of public functions typically forms the public API of the code. In addition to public functions, it's quite common

for code to also have some number of *private* functions. These are only visible to code within the class (or unit of code). The following snippet demonstrates this distinction:

```
class MyClass {
    String publicFunction() { ... }
    private String privateFunction1 { ... }
    private String privateFunction2 { ... }
}
```

This is visible to code outside the class.

This is visible only to code within the class.

Private functions are implementation details and they're not something that code outside the class should be aware of or ever make direct use of. Sometimes it can seem tempting to make some of these private functions visible to the test code so they can be directly tested. But this is often not a good idea, as it can result in tests that are tightly coupled to implementation details and that don't test the things we ultimately care about.

### 11.2.1 Testing private functions is often a bad idea

In the previous section, we established that it's important to test all the behaviors of the MortgageAssessor class (repeated in the following listing). The public API of the mortgage assessor class is the `assess()` function. In addition to this publicly visible function, the class also has two private helper functions: `isEligibleForMortgage()` and `getMaxLoanAmount()`. These are not visible to any code outside of the class and are therefore implementation details.

#### Listing 11.5 Class with private helper functions

```
class MortgageAssessor {
    ...
    MortgageDecision assess(Customer customer) { ... }
    private static Boolean isEligibleForMortgage(Customer customer) { ... }
    private static MonetaryAmount getMaxLoanAmount(Customer customer) { ... }
}
```

Public API.

Private helper functions.

Let's concentrate on one of the behaviors of the MortgageAssessor class that we need to test: that a mortgage application is rejected if the customer has a bad credit rating. One common way in which engineers can end up testing the wrong thing is to conflate the desired end result with an intermediate implementation detail. If we look more closely at the MortgageAssessor class, then we see that the private `isEligibleForMortgage()` helper function returns false if the customer has a bad credit rating. This can make it tempting to make the `isEligibleForMortgage()` function visible to test code so it can be tested. The following listing shows what the class would look like if an engineer makes the `isEligibleForMortgage()` visible like this. By making the



`isEligibleForMortgage()` function publicly visible, it's visible to all other code (not just the test code). The engineer has added a “Visible only for testing” comment to warn other engineers not to call it from anything other than test code. But as we've seen already throughout this book, small print like this is very easily overlooked.

#### Listing 11.6 Private function made visible

```
class MortgageAssessor {
    private const Double MORTGAGE_MULTIPLIER = 10.0;

    MortgageDecision assess(Customer customer) {
        if (!isEligibleForMortgage(customer)) {
            return MortgageDecision.rejected();
        }
        return MortgageDecision.approve(getMaxLoanAmount(customer));
    }

    /** Visible only for testing */
    static Boolean isEligibleForMortgage(Customer customer) {
        return customer.hasGoodCreditRating() &&
            !customer.hasExistingMortgage() &&
            !customer.isBanned();
    }

    ...
}
```

**Public API.**

**Which helper functions are called is an implementation detail.**

**Made publicly visible only so it can be directly tested.**

After having made the `isEligibleForMortgage()` function visible, the engineer would then likely write a bunch of test cases that call it and test that it returns true or false in the correct scenarios. The following listing shows one such test case. It tests that `isEligibleForMortgage()` returns false if a customer has a bad credit rating. As we'll see in a moment, there are a number of reasons why testing a private function like this can be a bad idea.

#### Listing 11.7 Testing a private function

```
testIsEligibleForMortgage_badCreditRating_ineligible() {
    Customer customer = new Customer(
        income: new MonetaryAmount(50000, Currency.USD),
        outgoings: new MonetaryAmount(25000, Currency.USD),
        hasGoodCreditRating: false,
        hasExistingMortgage: false,
        isBanned: false);

    assertThat(MortgageAssessor.isEligibleForMortgage(customer))
        .isFalse();
}
```

**Directly tests the “private” `isEligibleForMortgage()` function.**

The problem with making a private function visible and testing it like this is three-fold:

- The test is not actually testing the behavior we care about. We said a few moments ago that the outcome we care about is that a mortgage application is rejected if the customer has a bad credit rating. What the test case in listing 11.7 is actually testing is that there is a function called `isEligibleForMortgage()` that returns false when called with a customer with a bad credit rating. This doesn't guarantee that a mortgage application will ultimately be rejected in such a scenario. An engineer might inadvertently modify the `assess()` function to call `isEligibleForMortgage()` incorrectly (or to not call it at all). The test case in listing 11.7 would still pass, despite the `MortgageAssessor` class being badly broken.
- It makes the test non-agnostic to implementation details. The fact that there is a private function called `isEligibleForMortgage()` is an implementation detail. Engineers might want to refactor the code, for example, renaming this function or moving it to a separate helper class. Ideally any refactoring like that shouldn't cause any of the tests to fail. But because we're directly testing the `isEligibleForMortgage()` function, a refactoring like that will cause the tests to fail.
- We've effectively changed the public API of the `MortgageAssessor` class. A comment like "Visible only for testing" is very easily overlooked (it's small print in the coding contract). So, we might find that other engineers start calling the `isEligibleForMortgage()` function and relying on it. Before we know it we'll be unable to ever modify or refactor this function because so much other code is depending on it.

A good unit test should test the behaviors that ultimately matter. This maximizes the chance that the test will accurately detect breakages and it tends to keep the test agnostic to implementation details. These are two of the key features of a good unit test that were identified in chapter 10. Testing a private function often goes against both of these aims. As we'll see in the next two subsections, we can often avoid testing private functions by either testing via the public API or by ensuring that our code is broken into appropriate layers of abstraction.

### **11.2.2 Solution: prefer testing via the public API**

In the previous chapter we discussed the guiding principle of "test using only the public API." This principle aims to guide us toward testing the behaviors that actually matter and not implementation details. Whenever we find ourselves making an otherwise private function visible so that tests can call it, it's usually a red flag that we're breaking this guiding principle.

In the case of the `MortgageAssessor` class, the behavior that actually matters is that a mortgage application is rejected for a customer with a bad credit rating. We can test this behavior using only the public API by calling the `MortgageAssessor.assess()` function. The following listing shows how the test case might look if we did this. The test case now tests the behavior that matters rather than an implementation detail and we no longer need to make any of the otherwise private functions in the `MortgageAssessor` class visible.

**Listing 11.8 Testing via the public API**

```
testAssess_badCreditRating_mortgageRejected() {
    Customer customer = new Customer(
        income: new MonetaryAmount(50000, Currency.USD),
        outgoing: new MonetaryAmount(25000, Currency.USD),
        hasGoodCreditRating: false,
        hasExistingMortgage: false,
        isBanned: false);
    MortgageAssessor mortgageAssessor = new MortgageAssessor();

    MortgageDecision decision = mortgageAssessor.assess(customer);

    assertThat(decision.isApproved()).isFalse();
}
```

**Behavior tested via the public API.**

**NOTE****Be pragmatic**

Making a private function visible for testing, is almost always a red flag that implementation details are being tested, and there's usually a better alternative. But when applying the principle of "test using only the public API" to other things (such as dependencies), it's important to remember the advice in chapter 10 (section 10.3). The definition of the "public API" can be open to some amount of interpretation and some important behaviors (such as side effects) may fall outside of what engineers consider to be the public API. If a behavior is important and is something that we ultimately care about, then it should be tested, even if it's not part of what might be considered the public API.

For relatively simple classes (or units of code), it's often very easy to test all the behaviors using only the public API. Doing this results in better tests that will more accurately detect breakages and not be tied to implementation details. But when a class (or unit of code) is more complicated or contains a lot of logic, testing everything via the public API can start to get tricky. This is often a sign that the layer of abstraction is too thick and that the code might benefit from being split into smaller units.

**11.2.3 Solution: Split the code into smaller units**

In the previous two subsections, the logic for determining if a customer has a good credit rating was relatively simple: it just involved calling `customer.hasGoodCreditRating()`. So, it wasn't too difficult to fully test the `MortgageAssessor` class using only the public API. In reality, the temptation to make a private function visible for testing more often occurs when a private function involves more complicated logic.

To demonstrate this, imagine that determining whether a customer has a good credit rating involves calling an external service and processing the result. Listing 11.9 shows what the `MortgageAssessor` class might look like if this were the case. The logic for checking the customer's credit rating is now considerably more complicated, as noted by the following:

- The MortgageAssessor class now depends on CreditScoreService.
- The CreditScoreService service is queried with the customer ID in order to look up the customer's credit score.
- A call to the CreditScoreService can fail, so the code needs to handle this error scenario
- If the call succeeds, then the returned score is compared to a threshold to determine whether the customer's credit rating is good.

Testing all this complexity and all of these corner cases (such as error scenarios) via the public API now seems quite daunting and not at all easy. This is when engineers most often resort to making an otherwise private function visible in order to make the testing easier. In listing 11.9, the `isCreditRatingGood()` function has been made “visible only for testing” for this reason. This still incurs all the same problems that we saw earlier, but the solution of testing via the public API no longer seems so feasible due to how complicated the logic is. But as we'll see in a moment, there's a more fundamental problem here: the `MortgageAssessor` class is doing too much stuff.

#### Listing 11.9 More complicated credit rating check

```
class MortgageAssessor {
    private const Double MORTGAGE_MULTIPLIER = 10.0;
    private const Double GOOD_CREDIT_SCORE_THRESHOLD = 880.0;

    private final CreditScoreService creditScoreService; <-----
    ...

    MortgageDecision assess(Customer customer) {
        ...
    }

    private Result<Boolean, Error> isEligibleForMortgage(
        Customer customer) {
        if (customer.hasExistingMortgage() || customer.isBanned()) {
            return Result.ofValue(false);
        }
        return isCreditRatingGood(customer.getId());
    }

    /** Visible only for testing */ <-----
    Result<Boolean, Error> isCreditRatingGood(Int customerId) {
        CreditScoreResponse response = creditScoreService
            .query(customerId);
        if (response.errorOccurred()) {
            return Result.ofError(response.getError());
        }
        return Result.ofValue(
            response.getCreditScore() >= GOOD_CREDIT_SCORE_THRESHOLD);
    }
    ...
}
```

**The MortgageAssessor class depends on CreditScoreService.**

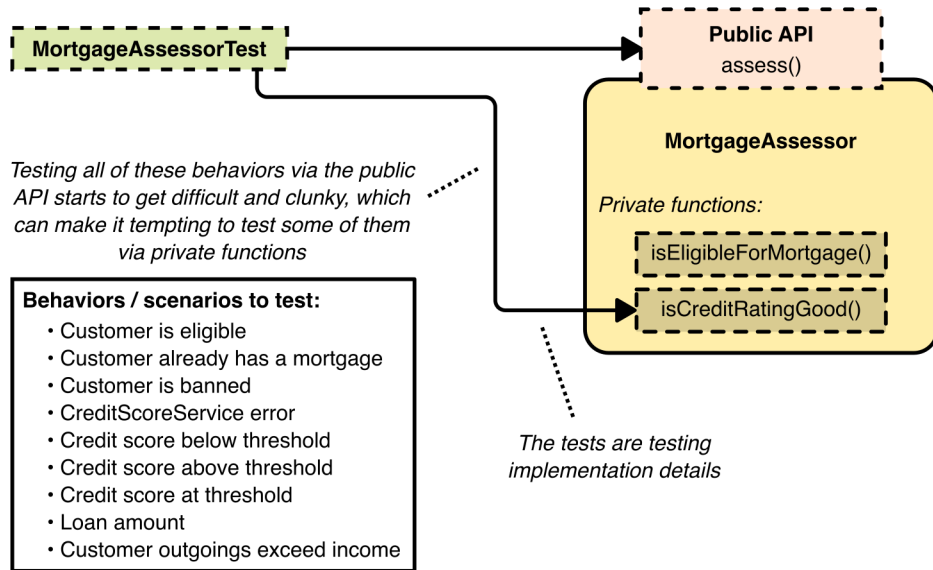
**isCreditRatingGood() function made visible for testing.**

**The CreditScoreService service is queried.**

**The score is compared to a threshold.**

**The error scenario of a call to the service failing is signalled via a Result type.**

Figure 11.1 illustrates the relationship between the test code (MortgageAssessorTest) and the code under test (MortgageAssessor).



**Figure 11.1** When a class does too much, it can be difficult to test everything using only the public API.

In chapter 2, when discussing layers of abstraction, we saw how it's often best not to place too many different concepts into a single class. The MortgageAssessor class contains a lot of different concepts, so in the language of chapter 2, the layer of abstraction it provides is “too thick.” This is the real reason why it seems hard to fully test everything using the public API.

The solution here is to split the code up into thinner layers. One way we might achieve this is to move the logic for determining if a customer has a good credit rating into a separate class. The following listing shows what this class might look like. The CreditRatingChecker class solves the subproblem of determining if a customer has a good credit rating. The MortgageAssessor class depends on CreditRatingChecker, meaning it's greatly simplified as it no longer contains all the nut-and-bolts logic for solving subproblems.

#### Listing 11.10 Code split into two classes

```
class CreditRatingChecker {
    private const Double GOOD_CREDIT_SCORE_THRESHOLD = 880.0;

    private final CreditScoreService creditScoreService;
    ...
}
```

A separate class to contain the logic for checking if a credit rating is good.

```

Result<Boolean, Error> isCreditRatingGood(Int customerId) {
    CreditScoreService response = creditScoreService
        .query(customerId);
    if (response.errorOccurred()) {
        return Result.ofError(response.getError());
    }
    return Result.ofValue(
        response.getCreditScore() >= GOOD_CREDIT_SCORE_THRESHOLD);
}

class MortgageAssessor {
    private const Double MORTGAGE_MULTIPLIER = 10.0;

    private final CreditRatingChecker creditRatingChecker; ←
    ...

    MortgageDecision assess(Customer customer) {
        ...
    }

    private Result<Boolean, Error> isEligibleForMortgage(
        Customer customer) {
        if (customer.hasExistingMortgage() || customer.isBanned()) {
            return Result.ofValue(false);
        }
        return creditRatingChecker ←
            .isCreditRatingGood(customer.getId()); ←
    }
    ...
}

```

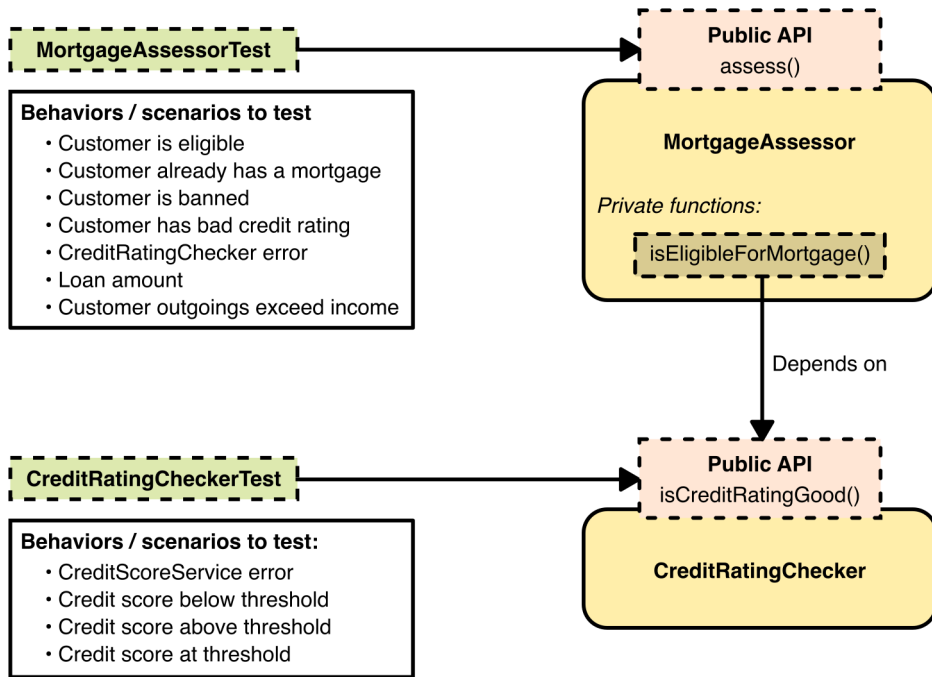
**MortgageAssessor  
depends on  
CreditRatingChecker.**

Both of the `MortgageAssessor` and `CreditRatingChecker` classes deal with a much more manageable number of concepts. This means that both of them can be easily tested using their respective public APIs, as shown in figure 11.2.

When we find ourselves making a private function visible so that we can test the code, it's usually a warning sign that we're not testing the behaviors that we actually care about. It's nearly always better to test the code using the already public functions. If this is not feasible, then it's often a sign that the class (or unit of code) is too big and that we should think about splitting it up into smaller classes (or units) that each solve a single subproblem.

### 11.3 Test one behavior at a time

As we've seen, there are often multiple behaviors that need to be tested for a given piece of code. In many cases each of these behaviors requires a slightly different scenario to be set up in order to test it. Meaning that the most natural thing to do is test each scenario (and its associated behavior) in its own test case. Sometimes, however, there may be a way to concoct a single scenario that tests multiple behaviors in one go. But just because this might be possible, it doesn't mean it's a good idea.



**Figure 11.2** Splitting a big class into smaller classes can make the code more testable.

### 11.3.1 Testing multiple behaviors at once can lead to poor tests

The following listing shows the code for a function to filter a list of coupons down to only the valid ones. The function takes a list of candidate coupons and returns another list containing only the ones that meet a set of criteria for being valid. There are a number of important behaviors that this function exhibits, as follows:

- Only valid coupons are returned.
- A coupon is considered invalid if it has already been redeemed.
- A coupon is considered invalid if it has expired.
- A coupon is considered invalid if it was issued to a different customer than the one given in the function call.
- The returned list of coupons is sorted in descending order of value.

#### **Listing 11.11** Code to get valid coupons

```
List<Coupon> getValidCoupons(
    List<Coupon> coupons, Customer customer) {
    return coupons
        .filter(coupon -> !coupon.alreadyRedeemed())
        .filter(coupon -> !coupon.hasExpired())
        .filter(coupon -> coupon.issuedTo() == customer)
        .sortBy(coupon -> coupon.getValue(), SortOrder.DESENDING);
}
```

As we’ve already discussed, it’s important that we test every behavior of a piece of code and the `getValidCoupons()` function is no exception to this. One approach we might be tempted to take is to write one massive test case that tests all the function behaviors in one go. The following listing shows what this might look like. The first thing to notice is that it’s quite hard to understand what exactly the test case is doing. The name `testGetValidCoupons_allBehaviors` is not very specific about what is being tested, and the amount of code in the test case makes it quite hard to follow. In chapter 10, we identified “understandable test code” as one of the key features of a good unit test. So straight away we can see that testing all the behaviors in one go like this, fails that criterion.

#### Listing 11.12 Testing everything at once

```
void testGetValidCoupons_allBehaviors() {
    Customer customer1 = new Customer("test customer 1");
    Customer customer2 = new Customer("test customer 2");
    Coupon redeemed = new Coupon(
        alreadyRedeemed: true, hasExpired: false,
        issuedTo: customer1, value: 100);
    Coupon expired = new Coupon(
        alreadyRedeemed: false, hasExpired: true,
        issuedTo: customer1, value: 100);
    Coupon issuedToSomeoneElse = new Coupon(
        alreadyRedeemed: false, hasExpired: false,
        issuedTo: customer2, value: 100);
    Coupon valid1 = new Coupon(
        alreadyRedeemed: false, hasExpired: false,
        issuedTo: customer1, value: 100);
    Coupon valid2 = new Coupon(
        alreadyRedeemed: false, hasExpired: false,
        issuedTo: customer1, value: 150);

    List<Coupon> validCoupons = getValidCoupons(
        [redeemed, expired, issuedToSomeoneElse, valid1, valid2],
        customer1);

    assertThat(validCoupons)
        .containsExactly(valid2, valid1)
        .inOrder();
}
```

Testing all the behaviors in one go also fails another of the criteria we identified in chapter 10: “well explained failures.” To understand why, let’s consider what happens if an engineer accidentally breaks one of the behaviors of the `getValidCoupons()` by removing the logic to check that a coupon has not already been redeemed. The `testGetValidCoupons_allBehaviors()` test case will fail, which is good (because the code is broken), but the failure message will not be particularly helpful at explaining which behavior has been broken (figure 11.3).



*Because the test case tests all the behaviors, we can't identify which behavior is broken from looking at the test case name*

```
Test case testGetValidCoupons_allBehaviors failed:
Expected:
[
  Coupon(redeemed: false, expired: false,
    issuedTo: test customer 1, value: 150),
  Coupon(redeemed: false, expired: false,
    issuedTo: test customer 1, value: 100)
]
But was actually:
[
  Coupon(redeemed: false, expired: false,
    issuedTo: test customer 1, value: 150),
  Coupon(redeemed: true, expired: false,
    issuedTo: test customer 1, value: 100),
  Coupon(redeemed: false, expired: false,
    issuedTo: test customer 1, value: 100)
]
```

*It's quite difficult to figure out which behavior is broken from the failure message*

**Figure 11.3** Testing multiple behaviors in one go can result in poorly explained test failures.

Having test code that's hard to understand and failures that are ill-explained not only wastes other engineers' time, but it can also increase the chance of bugs. As was discussed in chapter 10, if any engineer is intentionally changing one of the behaviors of the code, then we want to be sure that the other, seemingly unrelated behaviors are not accidentally affected too. A single test case that tests everything in one go tends to only tell us that something has changed, not exactly what has changed. So, it's much harder to have confidence about exactly which behaviors an intentional change has and hasn't affected.

### **11.3.2 Solution: Test each behavior in its own test case**

A much better approach is to test each behavior separately using a dedicated, well-named test case. The following listing shows what the test code might look like if we did this. We can see that the code inside each test case is now a lot simpler and easier to understand. We can identify from each test case name exactly which behavior is being tested, and it's relatively easy to follow the code to see how the test works. So, judging by the criterion that unit tests should have "understandable test code," the tests are now greatly improved.

**Listing 11.13 Testing one thing at a time**

```

void testGetValidCoupons_validCoupon_included() { <-----
    Customer customer = new Customer("test customer");
    Coupon valid = new Coupon(
        alreadyRedeemed: false, hasExpired: false,
        issuedTo: customer, value: 100);

    List<Coupon> validCoupons = getValidCoupons([valid], customer);

    assertThat(validCoupons).containsExactly(valid);
}

void testGetValidCoupons_alreadyRedeemed_excluded() { <-----
    Customer customer = new Customer("test customer");
    Coupon redeemed = new Coupon(
        alreadyRedeemed: true, hasExpired: false,
        issuedTo: customer, value: 100);

    List<Coupon> validCoupons =
        getValidCoupons([redeemed], customer);

    assertThat(validCoupons).isEmpty();
}

void testGetValidCoupons_expired_excluded() { ... <-----

void testGetValidCoupons_issuedToDifferentCustomer_excluded() { ... } <-----

void testGetValidCoupons_returnedInDescendingValueOrder() { ... } <-----

```

**Each behavior  
is tested in  
a dedicated  
test case.**

By testing each behavior separately and using an appropriate name for each test case, we now also achieve “well-explained failures.” Let’s again consider the scenario where an engineer accidentally breaks the `getValidCoupons()` function by removing the logic to check that a coupon has not already been redeemed. This will result in the `testGetValidCoupons_alreadyRedeemed_excluded()` test case failing. The name of this test case makes it clear exactly which behavior has been broken and the failure message (figure 11.4) is much easier to understand than the one we saw earlier.

*The name of the test case makes it immediately clear  
which behavior is broken*

```

Test case testGetValidCoupons_alreadyRedeemed_excluded failed:
Expected:
    []
But was actually:
    [
        Coupon(redeemed: true, expired: false,
                issuedTo: test customer, value: 100)
    ]

```

*The failure message is much easier to understand*

**Figure 11.4** Testing one behavior at a time often results in well explained test failures.

Despite the benefits of testing one thing at a time, writing a separate test case function for each behavior can sometimes lead to a lot of code duplication. This can seem especially clunky when the values and setup used in each test case are almost identical except for some minor differences. One way to reduce this amount of code duplication is to use *parameterized tests*. The next subsection explores this.

### 11.3.3 Parameterized tests

Some testing frameworks provide functionality for writing *parameterized tests*, this allows us to write a test case function once, but then run it multiple times with different sets of values in order to test different scenarios. The following listing shows how we might use a parameterized test to test two of the behaviors of the `getValidCoupons()` function. The test case function is marked with multiple `TestCase` attributes. Each of these defines two Booleans and a test name. The `testGetValidCoupons_excludesInvalidCoupons()` function has two Boolean function parameters, these correspond to the two Booleans defined in the `TestCase` attributes. When the tests run, the test case will be run once for each of the set of parameter values defined in the `TestCase` attributes.

**Listing 11.14 Parameterized test**

```
[TestCase(true, false, TestName = "alreadyRedeemed")]
[TestCase(false, true, TestName = "expired")]
void testGetValidCoupons_excludesInvalidCoupons(
    Boolean alreadyRedeemed, Boolean hasExpired) {
    Customer customer = new Customer("test customer");
    Coupon coupon = new Coupon(
        alreadyRedeemed: alreadyRedeemed,
        hasExpired: hasExpired,
        issuedTo: customer, value: 100);

    List<Coupon> validCoupons =
        getValidCoupons([coupon], customer);

    assertThat(validCoupons).isEmpty();
}
```

The parameter values are used during test setup.

The test case accepts different values via function parameters.

The test case will be run once with each set of parameter values.

#### NOTE

##### Ensure failures are well-explained

In listing 11.14 each set of parameters has an associated “TestName.” This ensures that any test failures are well-explained, because it will result in messages like “Test case testGetValidCoupons\_excludesInvalidCoupons.alreadyRedeemed failed.” (Notice that the test case name is suffixed with the name of the set of parameters that resulted in the failure: “alreadyRedeemed.”)

Adding names for each set of parameters is usually optional when writing parameterized tests. But omitting them can result in poorly explained test failures, so it’s good to think about what the test failures will look like when deciding if they’re needed or not.

Parameterized tests can be a great tool for ensuring that we test all the behaviors one at a time without repeating lots of code. The syntax and way in which parameterized tests are set up can vary a lot between different testing frameworks. Configuring parameterized tests can also be incredibly verbose and clunky in some frameworks and scenarios, so it's worth researching what the options are for whatever language you're using and considering the pros and cons. Some options are as follows:

- For C#, the NUnit test framework provides the `TestCase` attribute (similar to the example in listing 11.14): <https://docs.nunit.org/articles/nunit/writing-tests/attributes/testcase.html>
- For Java, JUnit provides support for parameterized tests: <https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests>
- For JavaScript, with the Jasmine test framework, it's relatively easy to write parameterized tests in a bespoke way, as described in this article: <https://medium.com/@nyablk97/parameterized-tests-with-jasmine-ecadb2856980>

### **11.4 Use shared test setup appropriately**

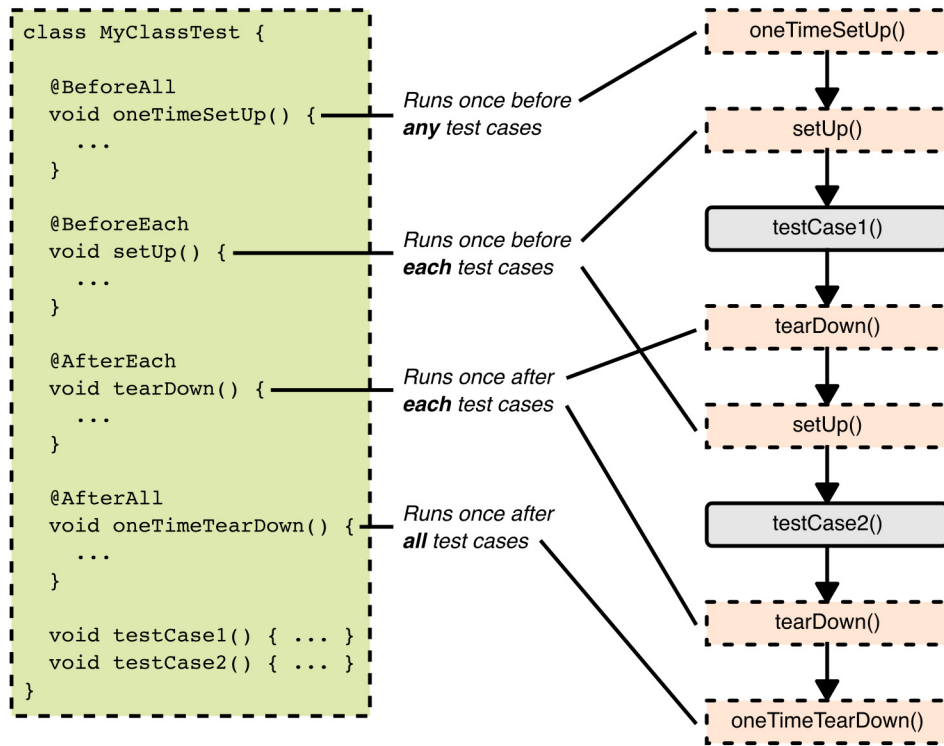
Test cases often require some amount of setup: constructing dependencies, population values in a test data store, or initializing other kinds of state. This setup can sometimes be quite laborious or computationally expensive and as such many testing frameworks provide functionality to try to make this easier. There are usually two distinct times at which setup code can be configured to run, distinguished by the following terms:

- **BeforeAll**—Setup code within a `BeforeAll` block will run once before any of the test cases are run. Some testing frameworks refer to this as `OneTimeSetUp` instead of using the term `BeforeAll`.
- **BeforeEach**—Setup code within a `BeforeEach` block will run once before each test case is run. Some testing frameworks refer to this as just `SetUp` instead of using the term `BeforeEach`.

In addition to providing ways to run setup code, frameworks also often provide ways to run teardown code. These are often useful for undoing any state that the setup code or test cases may have created. And again, there are usually two distinct times at which teardown code can be configured to run, distinguished by the following terms:

- **AfterAll**—Setup code within an `AfterAll` block will run once after all of the test cases have run. Some testing frameworks refer to this as `OneTimeTearDown` instead of using the term `AfterAll`.
- **AfterEach**—Teardown code within an `AfterEach` block will run once after each test case is run. Some testing frameworks refer to this as just `TearDown` instead of using the term `AfterEach`.

Figure 11.5 illustrates how these various pieces of setup and teardown might look in a piece of test code and the sequence in which they will run.



**Figure 11.5** Testing frameworks often provide a way to run setup and teardown code at various times relative to the test cases.

Using blocks of setup code like this results in setup being shared between different test cases. This can happen in two important, but distinct ways:

- *Sharing state*—If setup code is added to a `BeforeAll` block, then it will run once before all test cases. This means that any state it sets up will be shared between all the test cases. This type of setup can be useful when setup is slow or expensive (for example starting a test server or creating a test instance of a database). But if that state that gets set up is mutable, then there is a real risk that test cases might have adverse effects on one another (we'll explore this more in a moment).
- *Sharing configuration*—If setup code is added to a `BeforeEach` block, then it will run before each test case, meaning the test cases all share whatever configuration the code sets up. If that setup code contains a certain value or configures a dependency in a certain way, then each test case will run with that given value or a dependency configured in that way. Because the setup runs before each test case, there is no state shared between test cases. But as we'll see in a moment (in section 11.4.3), sharing configuration can still be problematic.

When setting up some particular state or dependency is expensive, using shared setup can be a necessity. Even when this isn't the case, shared setup can be a useful way to

simplify tests. If every test case requires a particular dependency, then it might be beneficial to configure it in a shared way rather than repeating a lot of boilerplate in every test case. But shared test setup can be a double-edged sword; using it in the wrong ways can lead to fragile and ineffective tests.

### 11.4.1 *Shared state can be problematic*

As a general rule, test cases should be isolated from one another. So, any actions that one test case performs should not affect the outcome of other test cases. Sharing mutable state between test cases makes it very easy to inadvertently break this rule.

To demonstrate this, listing 11.15 shows part of a class and function for processing an order. The two behaviors that we'll concentrate on are the following:

- If an order contains an out-of-stock item, then the order ID will be marked as delayed in the database.
- If payment for an order is not yet complete, then the order ID will be marked as delayed in the database.

#### **Listing 11.15** Code that writes to a database

```
class OrderManager {
    private final Database database;
    ...

    void processOrder(Order order) {
        if (order.containsOutOfStockItem() ||
            !order.isPaymentComplete()) {
            database.setOrderStatus(
                order.getId(), OrderStatus.DELAYED);
        }
        ...
    }
}
```

The unit tests should contain a test case for each of these behaviors (these are shown in the following listing). The `OrderManager` class depends on the `Database` class, so our tests need to set one of these up. Unfortunately, creating an instance of `Database` is computationally expensive and slow, so we create one in a `BeforeAll` block. This means that the same instance of database is shared between all the test cases (meaning the test cases share state). Unfortunately, this also makes the tests ineffective. To understand why, consider the sequence of events that happens when the tests run:

- The `BeforeAll` block will set up the database.
- The `testProcessOrder_outOfStockItem_orderDelayed()` test case will run. This results in the order ID being marked as delayed in the database.
- The `testProcessOrder_paymentNotComplete_orderDelayed()` test case then runs. Anything that previous test cases put in the database is still there (because the state is shared). So, one of two things might happen:
  - The code under test is called, everything works correctly, and it marks the order ID as delayed. The test case passes.

- The code under test is called, but it's broken. It doesn't save anything to the database to mark the order ID as delayed. Because the code is broken, we'd hope that the test case fails. But it instead passes, because `database.getOrderStatus(orderId)` still returns `DELAYED`, because the previous test case saved that value to the database.

#### Listing 11.16 State shared between test cases

```
class OrderManagerTest {

    private Database database;

    @BeforeAll
    void oneTimeSetUp() {
        database = Database.createInstance();
        database.waitForReady();
    }

    void testProcessOrder_outOfStockItem_orderDelayed() {
        Int orderId = 12345;
        Order order = new Order(
            orderId: orderId,
            containsOutOfStockItem: true,
            isPaymentComplete: true);
        OrderManager orderManager = new OrderManager(database);

        orderManager.processOrder(order);

        assertThat(database.getOrderStatus(orderId))
            .isEqualTo(OrderStatus.DELAYED);
    }

    void testProcessOrder_paymentNotComplete_orderDelayed() {
        Int orderId = 12345;
        Order order = new Order(
            orderId: orderId,
            containsOutOfStockItem: false,
            isPaymentComplete: false);
        OrderManager orderManager = new OrderManager(database);

        orderManager.processOrder(order);

        assertThat(database.getOrderStatus(orderId))
            .isEqualTo(OrderStatus.DELAYED);
    }
    ...
}
```

**The same instance of database is shared between all test cases.**

**OrderManager constructed with the shared database.**

**Results in the order ID being marked as delayed in the database.**

**OrderManager constructed with the shared database.**

**May pass even if the code is broken, because the previous test case saved this value to the database.**

Sharing mutable state between different test cases can very easily lead to problems. If at all possible, it's usually best to avoid sharing state like this. But if it is necessary, then we need to be very careful to ensure that the changes that one test case makes to the state don't affect other test cases.

### 11.4.2 Solution: Avoid sharing state or reset it

The most obvious solution to the problem of sharing mutable state is to just not share it in the first place. In the case of the `OrderManagerTest`, it would be more ideal if we didn't share the same instance of `Database` between test cases. So, if setting up a `Database` is less slow than we thought, then we might consider creating a new instance for each test case (either within the test cases or using a `BeforeEach` block).

Another potential way to avoid sharing mutable state is to use a test double (as discussed in chapter 10). If the team that maintains the `Database` class has also created a `FakeDatabase` for use in testing, then we could make use of this. Creating a `FakeDatabase` is likely fast enough that we can create a new one for each test case, meaning no state is shared.

If creating an instance of `Database` really is prohibitively slow and expensive (and we can't use a fake), then sharing an instance of it between test cases might well be unavoidable. If this is the case, then we should be very careful to ensure that the state is reset between each test case. This can often be achieved using an `AfterEach` block within the test code. As mentioned previously, this will run after each test case, so we can use it to ensure that the state is always reset before the next test case runs. The following listing shows what the `OrderManagerTest` test might look like if we use an `AfterEach` block to reset the database between test cases.

#### Listing 11.17 State reset between test cases

```
class OrderManagerTest {

    private Database database;

    @BeforeAll
    void oneTimeSetUp() {
        database = Database.createInstance();
        database.waitForReady();
    }

    @AfterEach
    void tearDown() {
        database.reset();
    }

    void testProcessOrder_outOfStockItem_orderDelayed() { ... }

    void testProcessOrder_paymentNotComplete_orderDelayed() { ... }

    ...
}
```

**The database is reset  
after each test case.**

**Test cases will never be affected by  
values saved by other test cases.**

#### NOTE

##### Global state

It's worth noting that the test code is not the only way in which state can be shared between test cases. If the code under test maintains any kind of global state, then we'll need to ensure that the test code resets this between test cases. Global state was discussed in chapter 9, and the conclusion was that it's usually best to avoid it. The impact that global state can have on the testability of code is yet another good reason for not using it.



Sharing mutable state between test cases is less than ideal. If it can be avoided, then this is usually preferable. If it can't be avoided, then we should ensure that the state is reset between each test case. This ensures that test cases don't have adverse effects on one another.

### 11.4.3 Shared configuration can be problematic

Sharing configuration between test cases doesn't immediately seem as dangerous as sharing state, but it can still result in ineffective tests. Imagine that another part of our infrastructure for processing orders is a system that generates postage labels for packages. The following listing contains the function that generates the data object to represent a postage label for an order. There are a few important behaviors that we need to test, but the one we will concentrate on is whether the package is marked as large. The logic for this is quite simple: if the order contains more than two items, then the package is considered large.

#### Listing 11.18 Postage label code

```
class OrderPostageManager {  
    ...  
  
    PostageLabel getPostageLabel(Order order) {  
        return new PostageLabel(  
            address: order.getCustomer().getAddress(),  
            isLargePackage: order.getItems().size() > 2,   
        );  
    }  
}
```

If the order contains more than two items, then the package is marked as large.

If we concentrate on just the `isLargePackage` behavior, then we need test cases for at least two different scenarios:

- An order containing two items. This should result in the package not being marked as large.
- An order containing three items. This should result in the package being marked as large.

If anyone inadvertently changes the logic in the code for deciding how many items makes a package large, then one of these test cases should fail.

Let's now imagine that constructing a valid instance of the `Order` class is more laborious than in previous subsections: we need to supply instances of the `Item` class, an instance of the `Customer` class, which also means creating an instance of the `Address` class. To save ourselves from repeating this configuration code in every test case, we decide to construct an instance of `Order` in a `BeforeEach` block (which runs once before each test case). The following listing shows what this looks like. The test case that tests the scenario where there are three items in the order uses the instance of `Order` created by the shared configuration. The `testGetPostageLabel_threeItems_largePackage()` test case therefore relies on the fact that the shared configuration creates an order containing exactly three items.

**Listing 11.19 Shared test configuration**

```

class OrderPostageManagerTest {
    private Order testOrder;

    @BeforeEach
    void setUp() {
        testOrder = new Order(
            customer: new Customer(
                address: new Address("Test address"),
            ),
            items: [
                new Item(name: "Test item 1"),
                new Item(name: "Test item 2"),
                new Item(name: "Test item 3"),
            ]);
    }
    ...

    void testGetPostageLabel_threeItems_largePackage() {
        PostageManager postageManager = new PostageManager();

        PostageLabel label =
            postageManager.getPostageLabel(testOrder);

        assertThat(label.isLargePackage()).isTrue();
    }
    ...
}

```

**Shared  
configuration**

**The test case relies on the fact that  
the shared configuration adds  
exactly three items to the order.**

This tests one of the behaviors that we care about and avoids the need to repeat a load of clunky code to create an `Order` in every test case. But unfortunately, things might go wrong if other engineers ever need to modify the tests. Imagine that another engineer now needs to add a new piece of functionality to the `getPostageLabel()` function: if any of the items in the order are hazardous, then the postage label needs to indicate that the package is hazardous. The engineer modifies the `getPostageLabel()` function to look like the following listing.

**Listing 11.20 A new piece of functionality**

```

class PostageManager {
    ...

    PostageLabel getPostageLabel(Order order) {
        return new PostageLabel(
            address: order.getCustomer().getAddress(),
            isLargePackage: order.getItems().size() > 2,
            isHazardous: containsHazardousItem(order.getItems());
        );
    }

    private static Boolean containsHazardousItem(List<Item> items) {
        return items.anyMatch(item -> item.isHazardous());
    }
}

```

**New functionality  
marks whether  
package is hazardous.**

The engineer has added a new behavior to the code, so they obviously need to add new test cases to test this. The engineer sees that there is an instance of `Order` constructed in the `BeforeEach` block and thinks “oh great, I can just add a hazardous item to that order and use that in one of my test cases.” The following listing shows the test code after they do this. This has helped the engineer test their new behavior, but they have inadvertently ruined the `testGetPostageLabel_threeItems_largePackage()` test case. The whole point in that test case is that it tests what happens when there are exactly three items in the order, but it’s now testing what happens when there are four items. So, the test no longer fully protects against the code being broken.

### Listing 11.21 Bad change to shared configuration

```
class OrderPostageManagerTest {
    private Order testOrder;

    @BeforeEach
    void setUp() {
        testOrder = new Order(
            customer: new Customer(
                address: new Address("Test address"),
            ),
            items: [
                new Item(name: "Test item 1"),
                new Item(name: "Test item 2"),
                new Item(name: "Test item 3"),
                new Item(name: "Hazardous item", isHazardous: true),
            ]
        );
    }
    ...

    void testGetPostageLabel_threeItems_largePackage() { ... }

    void testGetPostageLabel_hazardousItem_isHazardous() {
        PostageManager postageManager = new PostageManager();

        PostageLabel label =
            postageManager.getPostageLabel(testOrder);

        assertThat(label.isHazardous()).isTrue();
    }
    ...
}
```

**Fourth item added to order in shared configuration.**

**Now tests the case of four items, rather than intended case of three items.**

**New test case for testing that the label is marked with hazardous.**

### SHARED TEST CONSTANTS

`BeforeEach` or `BeforeAll` blocks are not the only ways to create shared test configuration. Using a shared test constant can often achieve exactly the same thing and can suffer the same set of potential problems we just discussed. If `OrderPostageManagerTest` configured the test order in a shared constant instead of a `BeforeEach` block, then it might look like the following snippet:

```

class OrderPostageManagerTest {
    private const Order TEST_ORDER = new Order(
        customer: new Customer(
            address: new Address("Test address"),
        ),
        items: [
            new Item(name: "Test item 1"),
            new Item(name: "Test item 2"),
            new Item(name: "Test item 3"),
            new Item(name: "Hazardous item", isHazardous: true),
        ]
    );
    ...
}

```

← A shared test constant

Technically, this also shares state between test cases, but it's good practice to only create constants using immutable data types, meaning no mutable state is shared. In this example, the `Order` class is immutable. If it were not immutable, then sharing an instance of `Order` in a shared constant would probably be even more of a bad idea (for the reasons discussed in section 11.4.1).

Shared configuration can be useful for preventing code repetition, but it's usually best not to use it to set up any values or state that specifically matter to test cases. It's very hard to keep track of exactly which test cases rely on which specific things in the shared configuration. And when changes are made in the future this can result in test cases no longer testing the thing they are intended to test.

#### 11.4.4 Solution: Define important configuration within test cases

It can seem laborious to repeat configuration in every test case, but when a test case relies on specific values or state being set up, it's often safer. And we can usually make this less laborious by using helper functions, so we don't have to repeat lots of boilerplate code.

In the case of testing the `getPostageLabel()` function, creating an instance of the `Order` class seemed quite clunky, but creating it in shared configuration resulted in the problems we saw in the previous subsection. We can mostly avoid both of these issues by defining a helper function for creating an instance of `Order`. Individual test cases can then call this function with the specific test values that they care about. This avoids lots of code repetition without having to use shared configuration and suffering the problems that can come with it. The following listing shows what the test code looks like with this approach.

##### Listing 11.22 Important configuration within test cases

```

class OrderPostageManagerTest {
    ...

    void testGetPostageLabel_threeItems_largePackage() {

```

```

Order order = createOrderWithItems([
    new Item(name: "Test item 1"),
    new Item(name: "Test item 2"),
    new Item(name: "Test item 3"),
]);
PostageManager postageManager = new PostageManager();

PostageLabel label = postageManager.getPostageLabel(order);

assertThat(label.isLargePackage()).isTrue();
}

void testGetPostageLabel_hazardousItem_isHazardous() {
    Order order = createOrderWithItems([
        new Item(name: "Hazardous item", isHazardous: true),
    ]);
    PostageManager postageManager = new PostageManager();

    PostageLabel label = postageManager.getPostageLabel(order);

    assertThat(label.isHazardous()).isTrue();
}
...

private static Order createOrderWithItems(List<Item> items) {
    return new Order(
        customer: new Customer(
            address: new Address("Test address"),
        ),
        items: items);
}
}

```

**Test cases perform  
their own setup for  
important things.**

**Helper function  
for creating an  
Order with  
specific items.**

When a piece of configuration directly matters to the outcome of a test case, it's usually best to keep it self-contained within that test case. This defends against future changes inadvertently ruining the tests, and it also makes the cause and effect within each test case clear (because everything that affects a test case in a meaningful way is there within the test case). Not every piece of configuration fits this description however, and the next subsection discusses when shared configuration can be a good idea.

#### 11.4.5 When shared configuration is appropriate

The previous subsections demonstrate why it's good to be cautious about the use of shared test configuration, but this doesn't mean that it's never a good idea to use it. Some pieces of configuration are necessary, but don't directly affect the outcome of test cases. In scenarios like this, using shared configuration can be an excellent way to keep tests focused and understandable by avoiding unnecessary code repetition and boilerplate.

To demonstrate this, imagine that constructing an instance of the `Order` class also requires providing some metadata about the order. The `PostageManager` class ignores this metadata, so it's completely irrelevant to the outcome of the test cases in `Order`-

PostageManagerTest. But it's still something that test cases need to configure, because an instance of the Order class can't be constructed without it. In a scenario like this, it makes a lot of sense to define the order metadata once as shared configuration. The following listing demonstrates this. An instance of OrderMetadata is placed into a shared constant called ORDER\_METADATA. Test cases can then use this constant instead of having to repeatedly construct this required, but otherwise irrelevant, data.

### Listing 11.23 Appropriate use of shared configuration

```
class OrderPostageManagerTest {
    private const OrderMetadata ORDER_METADATA =
        new OrderMetadata(
            timestamp: Instant.ofEpochSecond(0),
            serverIp: new IpAddress(0, 0, 0, 0));

    void testGetPostageLabel_threeItems_largePackage() { ... }
    void testGetPostageLabel_hazardousItem_isHazardous() { ... }
    ...

    void testGetPostageLabel_containsCustomerAddress() {
        Address address = new Address("Test customer address");
        Order order = new Order(
            metadata: ORDER_METADATA,
            customer: new Customer(
                address: address,
            ), items: []);

        PostageLabel label = postageManager.getPostageLabel(order);

        assertThat(label.getAddress()).isEqualTo(address);
    }
    ...

    private static Order createOrderWithItems(List<Item> items) {
        return new Order(
            metadata: ORDER_METADATA,
            customer: new Customer(
                address: new Address("Test address"),
            ),
            items: items);
    }
}
```

An instance of OrderMetadata is created in a shared constant.

Shared OrderMetadata used in test cases.

#### NOTE

#### Functions should ideally take only what they need

Chapter 9 discussed how function parameters should ideally be focused, meaning that functions take only what they need. If the tests for a piece of code require configuring a lot of values that are required but otherwise irrelevant to the behavior of the code, then it might be a sign that the function (or constructor) parameters are not focused enough. For example, we might argue that the `PostageManager.getPostageLabel()` function should take just an instance of `Address` and a list of items instead of a complete instance of the `Order` class. If this were the case, then the tests would not need to create irrelevant things like an instance of `OrderMetadata`.

Shared test setup can be a bit of a double-edged sword. It can be very useful for preventing code repetition or repeatedly performing expensive setup, but it also runs the risk of making tests ineffective or flakey. It's worth thinking carefully to ensure that it's used in an appropriate way.

## 11.5 Use appropriate assertion matchers

An *assertion matcher* is usually the thing in a test case that ultimately decides if the test has passed or not. The following snippet contains two examples of assertion matchers (`isEqualTo()` and `contains()`):

```
assertThat(someValue).isEqualTo(true);
assertThat(someList).contains("expected value");
```

If a test case fails, then the assertion matcher is also the thing that produces the failure message to explain why. Different assertion matchers produce different failure messages (depending on what they assert). In chapter 10, we identified “well-explained failures” as one of the key features of a good unit test. So, ensuring that we chose the most appropriate assertion matcher is important.

### 11.5.1 Inappropriate matchers can lead to poorly explained failures

To demonstrate how the use of an inappropriate matcher can lead to poorly explained test failures, we'll concentrate on testing the code in the following listing. `TextWidget` is a component used in a web app UI to display text. In order to control the styling of the component, various class names can be added to it. Some of these class names are hard-coded and other custom ones can be supplied via the constructor. The `getClassNames()` function returns a combined list of all class names. An important detail to note is that the documentation for the `getClassNames()` function states that the order of the returned class names is not guaranteed.

#### Listing 11.24 TextWidget code

```
class TextWidget {
    private const ImmutableList<String> STANDARD_CLASS_NAMES = | Hard-coded class names
        ["text-widget", "selectable"];
    private final ImmutableList<String> customClassNames;

    TextWidget(ImmutableList<String> customClassNames) { <-- Custom class names
        this.customClassNames = customClassNames;           supplied via the constructor.
    }

    /**
     * The class names for the component. The order of the class
     * names within the returned list is not guaranteed.
     */
    ImmutableList<String> getClassNames() { <-- Gets a list of all class
        return STANDARD_CLASS_NAMES.join(customClassNames); names (hard-coded
    }                                     and custom).
    ...
}
```

As we saw earlier, we should ideally aim to test one behavior at a time. One of the behaviors that we need to test is that the list returned by `getClassNames()` contains the `customClassNames`. One approach we might be tempted to take to test this, is to compare the returned list with an expected list of values. The following list shows this. But there are a couple of problems with this approach, as follows:

- The test case is testing more than it's meant to. The name of the test case suggests that it's only testing that the result contains the custom class names. But it's in fact also testing that the result contains the standard class names.
- If the order in which the class names are returned ever changes, then this test will fail. The documentation for the `getClassNames()` function explicitly says that the order is not guaranteed, so we should not create a test that fails when it changes. This could lead to false alarms or flakey tests.

#### Listing 11.25 Over-constrained test assertion

```
void testGetClassNames_containsCustomClassNames() {
    TextWidget textWidget = new TextWidget(
        ["custom_class_1", "custom_class_2"]);

    assertThat(textWidget.getClassNames()).isEqualTo([
        "text-widget",
        "selectable",
        "custom_class_1",
        "custom_class_2",
    ]);
}
```

Let's consider another idea we might try. Instead of comparing the returned result to an expected list, we might instead just check that the returned list contains the two values we care about "class\_1" and "class\_2." The following listing shows one way we might achieve this: by asserting that `result.contains(...)` returns true. This has solved the two problems we just saw: the test now only tests what it's meant to and a change in order will not cause the test to fail. But we've introduced another problem: the test failures will not be well explained (figure 11.6).

#### Listing 11.26 Test assertion with poor explainability

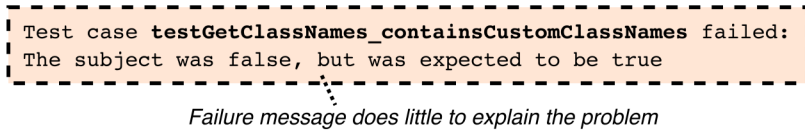
```
void testGetClassNames_containsCustomClassNames() {
    TextWidget textWidget = new TextWidget(
        ["custom_class_1", "custom_class_2"]);

    ImmutableList<String> result = textWidget.getClassNames();

    assertThat(result.contains("custom_class_1")).isTrue();
    assertThat(result.contains("custom_class_2")).isTrue();
}
```



Figure 11.6 shows what the failure message looks like if the test case fails due to one of the custom classes being absent. It's not obvious from this failure message how the actual result differs from the expected one.



Test case `testGetClassNames_containsCustomClassNames` failed:  
The subject was false, but was expected to be true

*Failure message does little to explain the problem*

**Figure 11.6** An inappropriate assertion matcher can result in a poorly explained test failure.

Making sure that a test fails when something is broken is essential, but as we saw in chapter 10, it's not the only consideration. We also want to ensure that a test only fails when something is genuinely broken and that test failures will be well explained. To achieve all these aims, we need to choose an appropriate assertion matcher.

### 11.5.2 Solution: Use an appropriate matcher

Most modern test assertion tools contain myriad different matchers than can be used in tests. One matcher on offer might be one that allows us to assert that a list contains at least a certain set of items in an unspecified order. Examples of such matchers are as follows:

- *In Java*—The `containsAtLeast()` matcher from the Truth library (<https://truth.dev/>).
- *In JavaScript*—The `jasmine.arrayContaining()` matcher from the Jasmine framework (<https://jasmine.github.io/>).

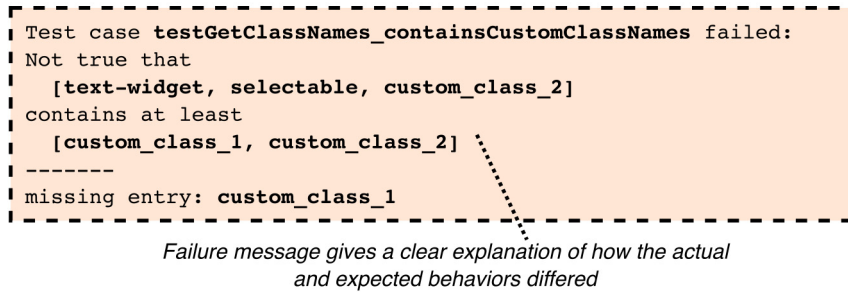
The following listing shows what our test case looks like if we use a `containsAtLeast()` matcher. The test case will fail if `getClassNames()` fails to return any of the custom class names. But it will not fail due to changes in other behaviors, such as the hard-coded class names being updated, or the order changing.

#### Listing 11.27 Appropriate assertion matcher

```
testGetClassNames_containsCustomClassNames() {
  TextWidget textWidget = new TextWidget(
    ["custom_class_1", "custom_class_2"]);

  assertThat(textWidget.getClassNames())
    .containsAtLeast("custom_class_1", "custom_class_2");
}
```

If the test case fails, then the failure message will be well-explained, as shown in figure 11.7.



**Figure 11.7** An appropriate assertion matcher will produce a well explained test failure.

In addition to producing better-explained failures, using an appropriate matcher often makes the test code slightly easier to understand. In the following snippet, the first line of code reads more like a real sentence than the second line:

```
assertThat(someList).contains("expected value");
assertThat(someList.contains("expected value")).isTrue();
```

In addition to ensuring that a test fails when the code is broken, it's important to think about how a test will fail. Using an appropriate assertion matcher can often make the difference between a well-explained test failure and a poorly explained one that will leave other engineers scratching their heads.

## 11.6 Use dependency injection to aid testability

Chapters 2, 8, and 9 provided examples where the use of dependency injection improved code. In addition to those examples, there's another very good reason to use dependency injection: it can make code considerably more testable.

In the previous chapter, we saw how tests often need to interact with some of the dependencies of the code under test. This occurs whenever a test needs to set up some initial values in a dependency or verify that a side effect has occurred in one. In addition to this, section 10.4 (chapter 10) explained how it's sometimes necessary to use a test double as a substitute for a real dependency. It's therefore clear that there are scenarios where a test will need to provide a specific instance of a dependency to the code under test. If there's no way for the test code to do this, then it might well be impossible to test certain behaviors.

### 11.6.1 Hard-coded dependencies can make code impossible to test

To demonstrate this, the following listing shows a class for sending invoice reminders to customers. The `InvoiceReminder` class doesn't use dependency injection and instead creates its own dependencies in its constructor. The `AddressBook` dependency is used by the class to lookup customers' email addresses and the `EmailSender` dependency is used to send emails.

**Listing 11.28 Class without dependency injection**

```

class InvoiceReminder {
    private final AddressBook addressBook;
    private final EmailSender emailSender;

    InvoiceReminder() {
        this.addressBook = DataStore.getAddressBook();
        this.emailSender = new EmailSenderImpl();
    }

    Boolean sendReminder(Invoice invoice) {
        EmailAddress? address =
            addressBook.lookupEmailAddress(invoice.getCustomerId());
        if (address == null) {
            return false;
        }
        return emailSender.send(
            address,
            InvoiceReminderTemplate.generate(invoice));
    }
}

```

**Dependencies are created in the constructor.**

**Email address looked up using addressBook.**

**Email sent using emailSender.**

There are a few behaviors that this class exhibits (such as the following) and we should ideally write a test case to test each of them:

- That the `sendReminder()` function sends an email to a customer when their address is in the address book.
- That the `sendReminder()` function returns true when an email reminder is sent.
- That the `sendReminder()` function does not send an email when the customer's email address cannot be found.
- That the `sendReminder()` function returns false when an email reminder is not sent.

Unfortunately, it's quite difficult (and maybe even impossible) to test all these behaviors with the class in its current form, for the following reasons:

- The class constructs its own `AddressBook` by calling `DataStore.getAddressBook()`. When the code runs in real life, this creates an `AddressBook` that connects to the customer database to lookup contact information. But it's not suitable to use this in tests, because using real customer data could well lead to flakiness as the data changes over time. Another, more fundamental, problem is that the environment the test runs in probably doesn't have permission to access the real database, so during testing the returned `AddressBook` might not even work.
- The class constructs its own `EmailSenderImpl`. This means the test will have the real-world consequence of sending real emails. This is not a side effect that a test should be causing and is an example where we need to protect the outside world from the test (as discussed in chapter 10).

Normally, an easy solution to both of these problems would be to use a test double for the `AddressBook` and the `EmailSender`. But in this scenario, we can't do this, because we have no way to construct an instance of the `InvoiceReminder` class with test doubles instead of real dependencies. The `InvoiceReminder` class has poor testability, and a likely consequence of this is that not all its behaviors will be tested properly, which obviously increases the chance of bugs in the code.

### 11.6.2 Solution: Use dependency injection

We can make the `InvoiceReminder` class a lot more testable and solve this problem by using dependency injection. The following listing shows what the class looks like if we modify it so its dependencies can be injected via the constructor. The class also includes a static factory function so it's still easy for real users of the class to construct it without having to worry about dependencies.

#### Listing 11.29 Class with dependency injection

```
class InvoiceReminder {
    private final AddressBook addressBook;
    private final EmailSender emailSender;

    InvoiceReminder(
        AddressBook addressBook,
        EmailSender emailSender) {
        this.addressBook = addressBook;
        this.emailSender = emailSender;
    }

    static InvoiceReminder create() {
        return new InvoiceReminder(
            DataStore.getAddressBook(),
            new EmailSenderImpl());
    }

    Boolean sendReminder(Invoice invoice) {
        EmailAddress? address =
            addressBook.lookupEmailAddress(invoice.getCustomerId());
        if (address == null) {
            return false;
        }
        return emailSender.send(
            address,
            InvoiceReminderTemplate.generate(invoice));
    }
}
```

**Dependencies injected  
via the constructor.**

**Static factory function.**

It's now very easy for tests to construct the `InvoiceReminder` class using test doubles (in this case a `FakeAddressBook` and a `FakeEmailSender`):

```
...
FakeAddressBook addressBook = new FakeAddressBook();
fakeAddressBook.addEntry(
    customerId: 123456,
    emailAddress: "test@test.com");
```

```
FakeEmailSender emailSender = new FakeEmailSender();

InvoiceReminder invoiceReminder =
    new InvoiceReminder(addressBook, emailSender);
...
```

As was mentioned in chapter 1, testability is heavily related to modularity. When different pieces of code are loosely coupled and reconfigurable, it tends to be much easier to test them. Dependency injection is an effective technique for making code more modular and as such it's also an effective technique for making code more testable.

## 11.7 Some final words on testing

Software testing is a massive topic, and the things we've covered in these final two chapters are just the tip of a much bigger iceberg. These chapters have looked at unit testing, which is the level of testing that engineers usually encounter most frequently in their everyday work. As was discussed in chapter 1, two other levels of testing that you're very likely to come across (and make use of) are:

- *Integration tests*—A system is usually built up of multiple components, modules, or subsystems. The process of linking these components and subsystems together is known as *integration*. Integration tests try to ensure that these integrations work and stay working.
- *End-to-end (E2E) tests*—These test typical journeys (or workflows) through a whole software system from start to finish. If the software in question were an online shopping store, then an example of an E2E test might be one that automatically drives a web browser to ensure that a user can go through the workflow of completing a purchase.

In addition to different levels of testing, there are many different types of testing. The definitions of these can sometimes overlap, and engineers don't always agree on exactly what they mean. A by-no-means-exhaustive list of a few concepts that it's good to be aware of are as follows:

- *Regression testing*—Tests that are regularly run in order to ensure that the behavior or functionality of the software has not changed in an undesirable way. Unit tests are usually an important part of regression testing, but it can also include other levels of testing, such as integration tests.
- *Golden testing*—Sometimes referred to as “characterization testing.” These are usually based on a saved snapshot of the output from the code for a given set of inputs. If the observed output of the code ever changes, then the tests will fail. These can be useful for ensuring that nothing has changed, but when the tests do fail, it can be difficult to determine the reason for the failure. These tests can also be incredibly fragile and flakey in some scenarios.
- *Fuzz testing*—This was discussed in chapter 3. Fuzz tests call the code with lots of random or “interesting” inputs and check that none of them cause the code to crash.

There is a wide and varied array of techniques that engineers can use to test software. Writing and maintaining software to a high standard often requires using a mixture of them. Although unit testing is probably the type of testing you will come across most, it's unlikely that it alone will fulfil all your testing needs. So, it's well worth reading about different types and levels of testing and keeping up to date with any new tools and techniques.

## **Summary**

- Concentrating on testing each function can easily lead to insufficient testing. It's usually more effective to identify all important behaviors and write a test case for each.
- Test the behaviors of the code that ultimately matter. Testing private functions is nearly always an indication that we're not testing the things that ultimately matter.
- Testing one thing at a time results in tests that are easier to understand as well as better explained test failures.
- Shared test setup can be a double-edged sword. It can avoid the repetition of code or expensive setup, but it can also lead to ineffective or flakey tests when used inappropriately.
- The use of dependency injection can considerably increase the testability of code.
- Unit testing is the level of testing that engineers tend to deal with most often. But it's by no means the only one. Writing and maintaining software to a high standard often requires the use of multiple testing techniques.

So, you've made it to the end (and even read the chapters about testing)! I hope you've enjoyed the journey through this book and learned some useful things along the way. Now we're done with the 11 chapters of prelude, onto the most important part of the book . . . the readable version of that chocolate brownie recipe in appendix A.

Chapter 7 from *Five Lines of Code*  
by Christian Clausen

**I**n this chapter, we learn how the compiler, with all its irksome complaints, can be an indispensable productivity tool if we take the time to get to know and trust it.



# *Work with the compiler*

---

In part 2, we look deeper at how to bring the rules and refactoring patterns into the real world. We dive into practices that enable us to take full advantage of the tools now at our disposal and discuss how they came to be as they are.

## ***This chapter covers:***

- Understanding the strengths and weaknesses of compilers
- Using the strengths to eliminate invariants
- Sharing responsibility with the compiler

When we are just learning to program, the compiler can feel like an endless source of nagging and nitpicking. It takes things too literally, it gives no leeway, and it freaks out over even the tiniest slip-ups. But used correctly, the compiler is one of the most important elements of our daily work. Not only does it transform our code from a high-level language to a lower-level one, but it also validates several properties and guarantees that certain errors will not occur when we run our program.

In this chapter, we start getting to know our compiler so we can actively use it and build on its strengths. Similarly, we will learn what it cannot do so we do not build on a weak foundation.



When we are intimately familiar with the compiler, we should make it part of our team by sharing the responsibility for correctness with it, letting it help build the software right. If we fight the compiler or trip it up, we are accepting a higher risk of bugs in the future, usually with minimal benefit.

Once we have accepted sharing the responsibility, we must trust the compiler. We need to make an effort to keep dangerous invariants to a minimum, and we need to listen to the compiler's output—including its warnings.

The final stage of this journey is accepting that the compiler is better at predicting program behavior than we are. It is quite literally a robot; it does not fatigue even when dealing with hundreds of thousands of lines of code. It can validate properties that no human realistically could. It is a powerful tool, so we should use it!

## 7.1 **Getting to know the compiler**

There are more compilers in the world than I can count, and new ones are invented all the time. So instead of focusing on a specific compiler, we discuss properties that are common to most compilers, including the mainstream Java, C#, and TypeScript variety.

A compiler is a program. It is good at certain things, like consistency; contrary to common folklore, compiling more than once will not yield different results. Likewise, it is bad at certain things, like judgment; compilers follow the common idiom “When in doubt, ask.”

Fundamentally, the compiler's goal is to generate a program in some other language that is equivalent to our source program. But as a service, modern compilers also verify whether specific errors can occur during runtime. This chapter focuses on the latter.

Like most things in programming, we get the best understanding from practicing. We need a deep understanding of what our compiler can and cannot do and how it can be fooled. Therefore, I always have an experimental project ready, to check how the compiler deals with something. Can it guarantee that this is initialized? Can it tell me whether `x` can be `null` here?

In the following sections, we answer both these questions by detailing some of the most common strengths and weaknesses of modern compilers.

### 7.1.1 **Weakness: The halting problem limits compile-time knowledge**

The reason we cannot say exactly what will happen during runtime is called the *halting problem*. In a nutshell, it states that without running a program, we cannot know how the program will behave—and even then, we observe only one path through our program.

**THE HALTING PROBLEM** In general, programs are fundamentally unpredictable.

For a quick demonstration of why this is true, consider the following program shown in the listing.

**Listing 7.1 Program without runtime errors**

```
if (new Date().getDay() === 35)
  5.foo();
```

We know that `getDay` will never return 35. So, whatever is inside the **if** will never be run and thus doesn't matter, even though it would fail because there is no method `foo` defined on the number 5.

Some programs will definitely fail and will be rejected. Some will definitely not fail and will be allowed. The halting problem means compilers have to decide how to deal with the programs in between. Sometimes the compiler allows programs that might not behave as expected, including failing during runtime. Other times it disallows a program if it cannot guarantee the program is safe; this is called a *conservative* analysis.

Conservative analyses prove that there is no possibility of some specific failure in our program. We can only rely conservative analyses.

Note that the halting problem is not specific to any compiler or language, it is an inherent property of programming languages. In fact, being subject to the halting problem is the very definition of being a programming language. Where languages and compilers differ is in when they are conservative and when not.

**7.1.2 Strength: Reachability ensures methods return**

One of the conservative analyses checks whether a method **returns** in every path. We are not allowed to run off the end of a method without hitting a **return** statement.

In TypeScript, it is legal to run off the end of a method; but if we use the method `assertExhausted` from chapter 4, we can get the desired behavior. Although the following listing looks like a runtime error, the `never` keyword forces the compiler to analyze whether there is any possible way to reach `assertExhausted`. In this example, the compiler figures out that we have not checked all values of the enum.

**Listing 7.2 Compiler error due to reachability**

```
enum Color {
  RED, GREEN, BLUE
}
function assertExhausted(x: never): never {
  throw new Error("Unexpected object: " + x);
}
function handle(t: Color) {
  if (t === Color.RED) return "#ff0000";
  if (t === Color.GREEN) return "#00ff00";
  assertExhausted(t);
}
```

The compiler errors  
because we have not  
handled `Color.BLUE`

We used this particular check to verify that our **switch** covered all cases in section 4.2.3. This is called an *exhaustiveness check* in typed functional languages, where it is much more common.

In general, this is a challenging analysis to take advantage of—especially when we follow the five-lines rule, since then it is easy to spot how many **returns** we have and where they are.

### 7.1.3 Strength: Definite assignment prevents accessing uninitialized variables

Another property that compilers are good at verifying is whether variables have definitely been assigned values before they are used. Note that this does not mean they contain anything useful; but they have been explicitly assigned *something*.

This check applies to local variables, specifically in cases where we want to initialize locals inside an **if**. In this case, we run the risk of not having initialized the variable in all paths. Consider this code to find an element whose name is John. At the **return** statement, there is no guarantee that we will have initialized the `result` variable; thus, the compiler will not allow this program.

#### Listing 7.3 Uninitialized variable

```
let result;
for (let i = 0; i < arr.length; i++)
  if (arr[i].name === "John")
    result = arr[i];
return result;
```

We may know that in this code, `arr` definitely contains an element whose name is John. In this case, the compiler is overly cautious. The optimal way to deal with this is to teach the compiler what we know: that it will find an element named John.

We can teach the compiler by taking advantage of the other target of the definite assignment analysis: read-only (or **final**) fields. A read-only field is required to be initialized at the termination of the constructor; that means we need to assign it either in the constructor or at the declaration directly.

We can use this strictness to ensure that specific values exist. In the earlier example, we can wrap our array in a class with a read-only field for the object whose name is John. Thereby we even avoid having to iterate through the list. Making this change does, of course, mean that we have to alter how the list is created. Still, by making this change, we prevent anyone from ever causing the John-object to disappear unnoticed, thereby eliminating an invariant.

### 7.1.4 Strength: Access control helps encapsulate data

The compiler is also excellent at access control, which we use when we have encapsulated data. If we make a member private, we can be sure that it does not escape accidentally. We saw plenty of examples of how and why to use this technique in chapter 6, so we will not go into further detail here, except for clearing up a common misconception among junior programmers: **private** applies to the class, *not* the object. This means we can inspect another object's private members if it is of the same class.

If we have methods that are sensitive to invariants, we can protect them by making them private, as shown in the following listing.

**Listing 7.4 Compiler error due to access**

```

class Class {
  private sensitiveMethod() {
    // ...
  }
}
let c = new Class();
c.sensitiveMethod();

```

Compiler error here

**7.1.5 Strength: Type checking proves properties**

The final strength of compilers that I want to highlight is the strongest of them all: the type checker. The type checker is responsible for checking that variables and members exist, and we used this functionality whenever we renamed something to get errors in part 1 of the book. It was also the type checker that enabled **Enforce sequence**.

In this example, we have encoded a list data structure that cannot be empty because it can only be made up of one element or an element followed by a list.

**Listing 7.5 Compiler error due to types**

```

interface NonEmptyList<T> {
  head: T;
}
class Last<T> implements NonEmptyList<T> {
  constructor(public readonly head: T) { }
}
class Cons<T> implements NonEmptyList<T> {
  constructor(
    public readonly head: T,
    public readonly tail: NonEmptyList<T>) { }
}
function first<T>(xs: NonEmptyList<T>) {
  return xs.head;
}
first([]);

```

Type error

Contrary to common jargon, being strongly typed is not a binary property. Programming languages can be more or less strongly typed; it is a spectrum. The subset of TypeScript that we consider in this book limits its type strength to be equivalent to Java's and C#'s. This level of type strength is sufficient to teach the compiler complex properties like not being able to pop something off an empty stack. However, this requires some mastery of type theory. Several languages have even stronger type systems, the most interesting of which are as follows, in generally increasing order of strength:

- Borrowing types (Rust)
- Polymorphic type inference (OCaml and F#)
- Type classes (Haskell)

- Union and intersection types (TypeScript)
- Dependent types of (Coq and Agda)

In languages with a decent type checker, teaching it properties of our program is the highest level of security we can get. It equals using the most sophisticated static analyzers or proving the properties manually, which is much harder and more error-prone. Learning how to do this is out of the scope of this book but considering the strength of this analysis and the benefits to be gained, I hope I have piqued your interest enough for you to seek it out on your own.

### 7.1.6 Weakness: Dereferencing null crashes our application

At the other end of the spectrum is `null`. `null` is dangerous because it causes failure if we try to invoke methods on it. Some tools can detect some of these cases, but they can rarely detect all of them, which means we cannot rely on the tools blindly.

If we turn off TypeScript's strict `null` check, it behaves like other mainstream languages. In many modern languages, code like this is accepted, even though we can call it with `average(null)` and crash the program.

#### Listing 7.6 Potential null dereference, yet no compiler error

```
function average(arr: number[]) {  
  return sum(arr) / arr.length;  
}
```

The risk of runtime error means we should be extra careful when dealing with nullable variables. I like to say that if you cannot see a `null` check of a variable, then it probably is `null`. Better to check it one time too many than too few.

Some IDEs might tell us that a `null` check is redundant, and I know how much that semi-transparency or strike-through hurts the eyes. However, I urge you not to remove these checks unless you are absolutely sure that they are too expensive or will never catch an error.

### 7.1.7 Weakness: Arithmetic errors causes overflows or crashes

Something compilers usually do not check is the dreaded division (or modulo) by zero. A compiler does not even check whether something can overflow. These are called *arithmetic errors*. Dividing an integer by zero causes a program to crash; even worse, overflows silently cause programs to behave strangely.

Repeating the earlier example, even if we know our program does not call `average` with `null`, almost no compiler will spot the potential division by zero if we call it with an empty array.

#### Listing 7.7 Potential division by zero, yet no compiler error

```
function average(arr: number[]) {  
  return sum(arr) / arr.length;  
}
```

Because the compiler is not much help, we need to be very careful when doing arithmetic. Make sure the divisor cannot be zero and that adding or subtracting numbers that are not large enough to over- or underflow, or use some variation of `BigIntegers`.

### 7.1.8 **Weakness: Out-of-bounds errors crashes our application**

Yet another place where the compiler is in hot water is when we directly access data structures. When we attempt to access an index that is not within the bounds of the data structure, it causes an out-of-bounds error.

Imagine that we have a function to find the index of the first prime in an array. We can use the function to find the first prime, as shown in the following listing.

#### **Listing 7.8 Potential access out of bounds, yet no compiler error**

```
function firstPrime(arr: number[]) {  
    return arr[indexOfPrime(arr)];  
}
```

However, if there is no prime in the array, such a function will return `-1`, which causes an out-of-bounds error.

There are two solutions to circumvent this limitation. Either traverse the entire data structure, if there is a risk of not finding the element we expect; or use the approach from the earlier discussion of definite assignment to prove that the element definitely exists.

### 7.1.9 **Weakness: Infinite loops stall our application**

A completely different way our programs can fail is when nothing happens, and we are left staring at a blank screen as our program loops quietly. Compilers generally do not help with this kind of error.

In this example in the following listing, we want to detect whether we are inside a string. However, we erroneously forgot to pass the previous `quotePosition` to the second call to `indexOf`. If `s` contains a quote, this is an infinite loop, but the compiler does not see it.

#### **Listing 7.9 Potential infinite loop, yet no compiler error**

```
let insideQuote = false;  
let quotePosition = s.indexOf("\"");  
while(quotePosition >= 0) {  
    insideQuote = !insideQuote;  
    quotePosition = s.indexOf("\"");  
}
```

These issues are being reduced by transitioning away from **while** to **for** and then **foreach**, and recently to higher-level constructions such as `forEach` in TypeScript, stream operations in Java and `Linq` in C#.

### 7.1.10 Weakness: Deadlocks and race conditions cause unintended behavior

A final category of trouble comes from multi-threading. A sea of issues can arise from having multiple threads that share mutable data: race conditions, deadlocks, starvation, etc.

TypeScript does not support multiple threads, so I cannot write examples of these errors in TypeScript. However, I can demonstrate them using pseudo-code.

A race condition is the first problem we run into with threads. It happens when two or more threads compete to read and write a shared variable. What can happen is that the two threads read the same value before updating it. See the following listings.

#### Listing 7.10 Pseudo-code for race condition

```
class Counter implements Runnable {
  private static number = 0;
  run() {
    for (let i = 0; i < 10; i++)
      console.log(this.number++);
  }
}
let a = new Thread(new Counter());
let b = new Thread(new Counter());
a.start();
b.start();
```

#### Listing 7.11 Example output

```
1
2
3
4
5
5
7
8
...
```

To solve this issue, we introduce locks. Let's give each thread a lock in the following listing and check that the other thread's lock is indeed free before proceeding.

#### Listing 7.12 Pseudo-code for deadlock

```
class Counter implements Runnable {
  private static number = 0;
  constructor(
    private mine: Lock, private other: Lock) { }
  run() {
    for (let i = 0; i < 10; i++) {
      mine.lock();
      other.waitFor();
```

```

        console.log(this.number++);
        mine.free();
    }
}
}
let aLock = new Lock();
let bLock = new Lock();
let a = new Thread(new Counter(aLock, bLock));
let b = new Thread(new Counter(bLock, aLock));
a.start();
b.start();

```

### Listing 7.13 Example output

```

1
2
3
4

```



The problem we have just stumbled upon is called a *deadlock*, where both threads are locked, waiting for each other to unlock before continuing. A common metaphor for this is two people meeting at a door, and both insisting that the other should go through first.

We can expose a final category of multi-threading errors if we make the loops infinite and just print out which thread is running, as shown in the following listing.

### Listing 7.14 Pseudo-code for starvation

```

class Printer implements Runnable {
    constructor(private name: string,
        private mine: Lock, private other: Lock) { }
    run() {
        while(true) {
            other.waitFor();
            mine.lock();
            console.log(this.name);
            mine.free();
        }
    }
}
let aLock = new Lock();
let bLock = new Lock();
let a = new Thread(
    new Printer("A", aLock, bLock));
let b = new Thread(
    new Printer("B", bLock, aLock));
a.start();
b.start();

```



**Listing 7.15 Example output**

```

A
A
A
A

```

Continues forever.

The problem here is that B is never allowed to run. This is quite rare but technically possible. It is called *starvation*. The metaphor for this is a one-lane bridge where one side has to wait, but the stream of cars from the other side never stops.

Entire books have been written about how to manage these issues. The best advice I can give to help alleviate them is to avoid having multiple threads with shared mutable data whenever possible. Whether this happens by avoiding the “multiple” part, the “sharing” part, or the “mutable” part depends on the situation.

## 7.2 Using the compiler

Having gotten familiar with our compiler, it is time to include it. The compiler should be part of the development team. Knowing how it can help us, we should design our software to take advantage of its strengths and avoid its weaknesses. We certainly should not fight with or cheat the compiler.

People often draw similarities between software development and construction. But as Martin Fowler has noted on his blog, this is one of the most damaging metaphors in our field. Programming is not construction; it is communication, on multiple levels:

- We communicate with the computer when we tell it what to do.
- We communicate with other developers when they read our code.
- We communicate with the compiler whenever we ask it to read our code.

As such, programming has much more in common with literature. We acquire knowledge about the domain, form a model in our heads, and then codify this model as a code. A beautiful quote states:

*Data structures are algorithms frozen in time.*

—Someone whose name eludes me

Dan North has noted the similarity that programs are the development team’s collective knowledge of the domain frozen in time. It is a complete, unambiguous description of everything the developers believe is true about the domain. In this metaphor, the compiler is the editor who makes sure our text meets a certain quality.

### 7.2.1 Making the compiler work

As we have seen many times now, there are several ways to design with the compiler in mind, thereby taking full advantage of having it on the team. Here is a short list of some of the ways we have used the compiler in this book.

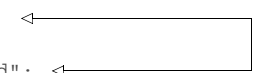
**GAIN SAFETY BY USING THE COMPILER AS A TODO LIST**

Probably the most common way we have taken advantage of the compiler in this book is as a todo list whenever we have broken something. When we want to change something, we simply rename the source method and rely on the compiler to tell us everywhere else we need to do something. This way, we are safe in the knowledge that the compiler does not miss any references. This works well, but only when we don't have other errors.

Imagine that we want to find every location where we use an enum to check whether we use **default**. We can find all usages of the enum, including those with a **default**, by appending something like `_handled` to the name. Now the compiler errors everywhere we use the enum. And once we have handled a location, we can simply append `_handled` to get rid of the error, as shown in the following listing.

**Listing 7.16 Finding enum usages with compiler errors**

```
enum Color_handled {
    RED, GREEN, BLUE
}
function toString(c: Color) {
    switch (c) {
        case Color.RED: return "Red";
        default: return "No color";
    }
}
```



Compiler errors.

Once we are finished, we can easily remove `_handled` everywhere.

**GAIN SAFETY BY ENFORCING SEQUENCES**

The pattern **Enforce sequence** is dedicated to teaching the compiler about an invariant in our program, thereby making the invariant a property, instead. This means the invariant can no longer accidentally be broken in the future, because the compiler guarantees that the property still holds every time we compile.

In chapter 6, we discussed both internal and external variants of using classes to enforce sequences. These classes both guarantee that a string has at some prior point been capitalized, as shown in the following listing.

**Listing 7.17 Internal**

```
class CapitalizedString {
    private value: string;
    constructor(str: string) {
        this.value = capitalize(str);
    }
    print() {
        console.log(this.value);
    }
}
```

**Listing 7.18 External**

```

class CapitalizedString {
  public readonly value: string; ←
  constructor(str: string) {
    this.value = capitalize(str);
  }
}

function print(str: CapitalizedString) {
  console.log(str.value);
}

```

**Method vs. function with a specific parameter type**

**GAIN SAFETY BY ENFORCING ENCAPSULATION**

By using the compiler's access control to enforce strict encapsulation, we localize our invariants. By encapsulating our data, we can be much more confident that it is kept in the shape we expect.

We already saw how to prevent someone from accidentally calling a helper method `depositHelper` by making it private, as shown in the following listing.

**Listing 7.19 Private helper**

```

class Transfer {
  constructor(from: string, private amount: number) {
    this.depositHelper(from, -this.amount);
  }
  private depositHelper(to: string, amount: number) {
    let accountId = database.find(to);
    database.updateOne(accountId, { $inc: { balance: amount } });
  }
  deposit(to: string) {
    this.depositHelper(to, this.amount);
  }
}

```

**GAIN SAFETY BY LETTING THE COMPILER DETECT UNUSED CODE**

We have also used the compiler to check whether code is unused with the refactoring pattern **Try delete then compile**. Deleting a flurry of methods at once, the compiler can quickly scan through our entire codebase and let us know which methods are used.

We use this approach to get rid of methods in interfaces. The compiler cannot know whether they are going to be used or are genuinely unused. But if we know that an interface is only used internally, we can simply try deleting methods from the interface and see if the compiler accepts the program.

In this code from chapter 4, we can safely delete both `m2` methods and even the `m3` method.

**Listing 7.20 Example with deletable methods**

```

interface A {
  m1(): void;
  m2(): void;
}

```

```
class B implements A {
    m1() { console.log("m1"); }
    m2() { m3(); }
    m3() { console.log("m3"); }
}
let a = new B();
a.m1();
```

### GAIN SAFETY WITH DEFINITE VALUES

Finally, earlier in this chapter, we showed a list data structure that could not be empty. We guarantee this by using read-only fields. These are in the compiler's definite assignment analysis and must have a value at the termination of the constructor. Even in a language that supports multiple constructors, we cannot end up with an object with uninitialized read-only fields.

#### Listing 7.21 Non-empty list due to read-only fields

```
interface NonEmptyList<T> {
    head: T;
}
class Last<T> implements NonEmptyList<T> {
    constructor(public readonly head: T) { }
}
class Cons<T> implements NonEmptyList<T> {
    constructor(
        public readonly head: T,
        public readonly tail: NonEmptyList<T>) { }
}
```

### 7.2.2 Don't fight the compiler

On the other hand, it saddens me every time I see someone deliberately fighting their compiler and preventing it from doing its part. There are several ways to do this; in the following, we give a short account of the most common. They happen primarily due to one of three offenses, each with a section dedicated to it: not understanding types, being lazy, and not understanding architecture.

#### TYPES

As described earlier, the type checker is the strongest part of the compiler. Therefore, tricking it or disabling it is the worst offense. There are three different ways people misuse the type checker.

#### CASTS

The first is using casts. A cast is like telling the compiler that you know better than it does. Casts prevent the compiler from helping you and essentially disable it for the particular variable or expression. Types are not intuitive; they are a skill that must be learned. Needing a cast is a symptom that either we don't understand the types, or someone else didn't.

We use a cast when a type is not what we need it to be. Using a cast is like giving a painkiller to someone in chronic pain: it helps right now but does nothing to fix the issue.

A common place for casts is when we have untyped JSON from a web service. In this example, the developer was confident that the JSON in the variable was always a number.

#### Listing 7.22 Cast

```
let num = <number> JSON.parse(variable);
```

There are two possible situations here: either we get the input from somewhere we have control over, such as our own web service, or—a more permanent solution—we reuse the same types on the sending side as the receiving side. Several libraries are available to assist with this. If the input comes from a third party, the safest solution is to parse the input with a custom parser.

#### DYNAMIC TYPES

Even worse than essentially disabling the type checker is *actually* disabling the type checker. This happens when we use dynamic types: in TypeScript, by using `any` (dynamic in C#). While this may seem useful, especially when sending JSON objects back and forth over HTTP, it opens up a myriad of potential errors, such as referring to fields that don't exist or that have different types than we expect, so that we end up attempting to multiply two strings.

I recently came across an issue where some TypeScript was running in version ES6 but the compiler was configured as ES5, meaning the compiler didn't know about all the methods in ES6. Specifically, it did not know `findIndex` on arrays. To solve this, a developer cast the variable to `any` so the compiler allowed any call on it.

#### Listing 7.23 Using any

```
(<any> arr).findIndex(x => x === 2);
```

It was unlikely that this method would not be present at runtime, so it was not too dangerous. However, updating the config would have been a safer and more permanent solution.

#### RUNTIME TYPES

The third way people fool the compiler is by moving knowledge from compile-time to runtime. This is the exact opposite of all the advice in this book. Here is a fairly common example of how it happens. Imagine that we have a method with 10 parameters. This is confusing, and every time we add or remove one, we need to correct it everywhere the method is called. So, instead of using 10 parameters, we decide to use only 1: a `Map` from strings to values. Then we can easily add more values to it without having to change any code. This is a horrible idea because we throw away knowledge. The compiler cannot know what keys exist in the `Map` and therefore cannot check whether we ever access a key that does not exist. We have moved from the strength of type checking to the weakness of out-of-bounds errors. Tired of laundry? Easy solution: burn all your clothes!

In this example listing, instead of passing three separate arguments, we pass one map. We can then pull out the values with `get`.

#### Listing 7.24 Runtime types

```
function stringConstructor(
  conf: Map<string, string>,
  parts: string[]) {
  return conf.get("prefix")
    + parts.join(conf.get("joiner"))
    + conf.get("postfix");
}
```

A safer solution is to make an object with those specific fields.

#### Listing 7.25 Static types

```
class Configuration {
  constructor(
    public readonly prefix: string,
    public readonly joiner: string,
    public readonly postfix: string) { }
}

function stringConstructor(
  conf: Configuration,
  parts: string[]) {
  return conf.prefix
    + parts.join(conf.joiner)
    + conf.postfix;
}
```

#### LAZINESS

The second great offense is laziness. I don't feel as though programmers are to blame for being lazy, since it is what got most people into programming in the first place. We happily spend hours or weeks tirelessly working to automate something we are too lazy to do. Being lazy makes us better programmers; staying lazy makes us worse programmers.

Another reason for my leniency in this offense is that developers are often under tremendous stress and tight deadlines to deliver. In that state of mind, everyone takes as many shortcuts as they can. The problem is, they are short-term fixes.

#### DEFAULTS

We discussed default values quite a bit in part 1. Wherever we use a default value, eventually someone will add a value that should not have the default and forget to correct it. Instead of using defaults, have the developer take responsibility every time they add or change something. This is done by not supplying a default value, so the compiler will force the developer to make the decision. This can even help to expose holes in the understanding of the problem to be solved when the compiler asks us a question, we do not know the answer to.

In this code, the developer wanted to take advantage of the fact that most people are female, so she made that the default. However, we can easily forget to override this, especially since we get no help from the compiler:

#### Listing 7.26 Bug due to default arguments

```
class Person {
  constructor(name: string, isFemale = true) { ... }
}
let john = new Person("John");
```

John is now female.

#### INHERITANCE

Through the rule **Only inherit from interfaces** and section 4.3 I have made my opinion of sharing code through inheritance abundantly clear, and my arguments as well. Inheritance is a form of default behavior and covered by the earlier section. Further, inheritance adds coupling between its implementing classes.

In this example, if we add another method to `Mammal`, we have to remember to manually check whether this method is valid in all descendent classes. It is easy to miss some or entirely forget to check. In this code, we have added a `laysEggs` method to the `Mammal` superclass, which works for most descendants—except `Platypus`.

#### Listing 7.27 Problem due to inheritance

```
class Mammal {
  laysEggs() { return false; }
}
class Dolphin extends Mammal { }
/// ...
class Platypus extends Mammal {
}
```

Should have overwritten `laysEggs`.

#### UNCHECKED EXCEPTIONS

Exceptions often come in two flavors: those we are forced to handle and those we are not. But if an exception can happen, we should handle it somewhere, or at least let the caller know that we have not handled it. This is exactly the behavior of checked exceptions. We should use unchecked exceptions only for things that cannot happen, such as when we know some invariant to be true, but we cannot express the invariant in the language. Having one unchecked exception called `Impossible` seems sufficient. But as with all invariants, we risk that one day it will be broken, and we will have an unhandled `Impossible` exception.

In this example, we can see the issue with using an unchecked exception for something that is not impossible. We reasonably check whether the input array is empty because that would cause an arithmetic error. However, because we use an unchecked exception, the caller can still call our method with an empty array, and the program will still crash.

**Listing 7.28 Using an unchecked exception**

```

class EmptyArray extends RuntimeException { }
function average(arr: number[]) {
  if (arr.length === 0) throw new EmptyArray();
  return sum(arr) / arr.length;
}
// ...
console.log(average([]));

```

A better solution is to use a checked exception. If a local invariant at the call site guarantees that the exception cannot happen, we can easily use the `Impossible` exception mentioned earlier. This is pseudo-code, as TypeScript, unfortunately, does not have checked exceptions, as shown in the following listing.

**Listing 7.29 sing an unchecked exception**

```

class Impossible extends RuntimeException { }
class EmptyArray extends CheckedException { }
function average(arr: number[]) throws EmptyArray {
  if (arr.length === 0) throw new EmptyArray();
  return sum(arr) / arr.length;
}
// ...
try {
  console.log(average(arr));
} catch (EmptyArray e) {
  throw new Impossible();
}

```

**ARCHITECTURE**

The third way people prevent the compiler from helping is due to a lack of understanding of the architecture: specifically, the micro-architecture. *Micro-architecture* is architecture that effects this team but not other teams.

We discussed the primary way this comes to fruition in part 1: breaking encapsulation with getters and setters. Doing so creates coupling between the receiver and the field and prevents the compiler from controlling the access.

In this stack implementation, we break encapsulation by exposing the internal array. This means external code can depend on it. Even worse, the external code can change the stack by changing the array, as shown in the following listing.

**Listing 7.30 Poor micro-architecture with a getter**

```

class Stack<T> {
  private data: T[];
  getArray() { return this.data; }
}
stack.getArray()[0] = newBottomElement;

```

This line changes the stack.

Another way this can happen is if we pass a private field as an argument, which has the same effect. In this example, the method that gets the array can do anything with it, including changing the stack. Never mind that the function is misleadingly named.



**Listing 7.31 Poor micro-architecture with a parameter**

```

class Stack<T> {
  private data: T[];
  printLast() { printFirst(this.data); }
}
function printFirst<T>(arr: T[]) {
  arr[0] = newBottomElement;
}

```

This line changes the stack.

Instead, we should pass **this**, so we can keep our invariants local.

## 7.3 Trusting the compiler

We now actively use the compiler and build software with it in mind. With our knowledge of its strengths and weaknesses, we rarely get into frustrating arguments with the compiler, and we can begin to trust it.

We can move away from the counterproductive feeling that we know better than the compiler and pay close attention to what it says. We get back what we put into it, and following the last section, we now put a lot into it.

Let's examine the two final frontiers where people tend to distrust the compiler: invariants and warnings.

### 7.3.1 Teach the compiler invariants

The malice of global invariants has been discussed at length throughout the book, so they should be under control by now. But what about local invariants?

Local invariants are easier to maintain because their scope is limited and explicit. However, they come with the same conflicts with the compiler. We know something about the program that our compiler does not.

Let's look at a larger example where this comes into play. Here, we are creating a data structure to count elements. Thus, when we add elements, the data structure keeps track of how many of each type of element we have added. For convenience, we also keep track of the total number of elements added, as shown in the following listing.

**Listing 7.32 Counting set**

```

class CountingSet {
  private data: StringMap<number> = {};
  private total = 0;
  add(element: string) {
    let c = this.data.get(element);
    if (c === undefined)
      c = 0;
    this.data.put(element, c + 1);
    this.total++;
  }
}

```

Keeping track of the total.

Keeping track of the total.

We want to add a method to pick a random element out of this data structure. We could do this by picking a random number less than the total and return the element

that would have been at that position if this were an array. Because we are not storing an array, we instead need to iterate through the keys and jump forward by that many places in the index, as shown in the following listing.

### Listing 7.33 Picking a random element (error)

```
class CountingSet {
  // ...
  randomElement(): string {
    let index = randomInt(this.total);
    for (let key in this.data.keys()) {
      index -= this.data[key];
      if (index <= 0)
        return key;
    }
  }
}
```

← Error due to reachability.

This method doesn't compile, since we fail the reachability analysis described earlier. The compiler does not know that we will always select an element because it does not know the invariant that `total` is the number of elements in the data structure. This is a local invariant, kept true at the termination of every method in this class.

In this case, we can resolve the error by adding an impossible exception, as shown in the following listing.

### Listing 7.34 Picking a random element (fixed)

```
class Impossible { }
class CountingSet {
  // ...
  randomElement(): string {
    let index = randomInt(this.total);
    for (let key in this.data.keys()) {
      index -= this.data[key];
      if (index <= 0)
        return key;
    }
    throw new Impossible();
  }
}
```

← Exception to avoid an error.

However, this only solves the immediate issue of the compiler complaining; we have not added any security that this invariant will not be broken later. Imagine implementing a `remove` function and forgetting to decrease `total`. The compiler dislikes our `randomElement` method because it is dangerous.

Whenever we have invariants in a program, we go through an adapted version of “If you can’t beat them, join them”:

- 1 Eliminate them.
- 2 If you can’t, then teach the compiler about them.
- 3 If you can’t, then teach the runtime about them with an automated test.

- 4 If you can't, then teach the team about them by documenting them extensively.
- 5 If you can't, then teach the tester about them and test them manually.
- 6 If you can't, then start praying because nothing earthbound can help you.

“Can’t” in this context means infeasible, rather than impossible. There is a time for each of these solutions. But note that the lower we go on the list, the longer we commit ourselves to maintaining the solution. Documentation requires more deliberate time to maintain than tests do, because tests tell you when they grow out of sync with the software; documentation offers no such courtesy. The higher each option is on the list, the cheaper it is in the long term. This should disarm the all-too-common excuse that we have no time to write tests, as not doing it is sure to be more time expensive in the long term.

Note that if your software has a short lifetime, you can permit yourself to select an option lower on the list: for example, if you are building a prototype that is to be thrown out after testing it manually.

### 7.3.2 Pay attention to warnings

The other area where people tend to distrust the compiler is when it gives warnings. In hospitals, there is a term called *alarm fatigue*: healthcare workers become desensitized to noises because they are the norm rather than the exception. The same effect happens in software: each time we ignore a warning, a runtime error, or a bug, we pay a little less attention to them in the future. Another perspective on warning fatigue is the broken window theory, which states that if something is in pristine condition, people strive to keep it that way, but as soon as something is bad, we are less reluctant to put something bad next to it.

Even if some warnings are unjustified, the danger is that we might miss a crucial one because the insignificant ones drown it out. This is one of the most critical dangers to understand. Insignificant errors or warnings can shadow more significant errors.

The fact of the matter is, the warnings are there for a reason: to help us make fewer mistakes. Therefore, only one number of warnings is healthy: zero. In some codebases, this seems impossible because warnings have run rampant for too long; in such situations, we set an upper limit on the number of warnings that we allow in the codebase and then decrease this number bit by bit every month. This is a daunting task, especially since we won’t reap any significant benefits in the beginning, before the number is low. Once we are at zero, we should enable the language configuration for disallowing warnings, to ensure that they never raise their ugly heads again.

## 7.4 Trusting the compiler exclusively

*Will this work?*

—Every programmer

The final stage of this journey is when we have a pristine codebase, where we listen to and trust the compiler and design with it in mind. At this stage, we are so intimately

familiar with its strengths and weaknesses that instead of having to trust our judgment, we can be satisfied with the compiler's. Instead of straining ourselves, wondering whether something will work, we can just ask the compiler.

If we have taught our compiler the structure of our domain, have encoded the invariants, and are used to warning-free output that we can trust, successful compiling should give us more confidence than we could have gotten simply from reading the code. Of course, the compiler cannot know whether our code solves the problem we expect it to, but it can tell us whether the program can crash, which is never what we expect.

This does not happen overnight. It requires lots of practice and discipline on the journey. It also requires the proper technologies (that is, programming language). This quote includes the compiler:

*If you're the smartest person in the room, you're in the wrong room.*

—Origin unknown

## Summary

- Know the common strengths and weaknesses of modern compilers. We can adjust our code to avoid the weaknesses and take advantage of the strengths.
  - Use reachability to ensure that switch covers all cases.
  - Use definite assignment to ensure variables have values.
  - Use access control to protect methods with sensitive invariants.
  - Check to make sure variables are not null before dereferencing them.
  - Check that numbers are not zero before dividing with them.
  - Check that operations will not over- or underflow or use `BigInteger`'s.
  - Avoid out-of-bounds errors by traversing the entire data structure or use definite assignment.
  - Avoid infinite loops by using higher-level constructions.
  - Avoid threading issues by not having multiple threads share mutable data.
- Learn to use the compiler instead of fighting it, to reach higher levels of safety.
  - Use compiler errors as a todo list when refactoring.
  - Use the compiler to enforce sequence invariants.
  - Use the compiler to detect unused code.
  - Don't use type casts, nor dynamic or runtime types.
  - Don't use defaults, inheritance from classes, or unchecked exceptions.
  - Pass **this** instead of private fields to avoid breaking encapsulation.
- Trust the compiler, value its output, and avoid warning fatigue by keeping a pristine codebase.
- Rely on the compiler to predict whether code will work.

**I**n this chapter, we'll look at testing from a different angle, discovering how it eases our workload and realizing the value of the assurances only it can give us.

# Tasty testing



## ***This chapter covers***

- Understanding why we hate testing and how we can love it
- Making testing more enjoyable
- Avoiding TDD, BDD, and other three-letter acronyms
- Deciding what to test
- Doing less work using tests
- Making tests spark joy

Many software developers would liken testing to writing a book: it's tedious, nobody likes doing it, and it rarely pays off. Testing is considered like a second-class activity compared to coding, not doing *the real work*. Testers are subjected to preconceptions like they are having it too easy.

The reason behind the dislike for testing is that we developers see it as a process disconnected from building software. Building software is all about writing code from a programmer's perspective, whereas it's all about setting the right course for the team from a manager's vantage point. Similarly, for a tester, it's all about the quality of the product. We consider testing an external activity because of our perception that it's not part of the software development and we want to get involved as little as possible.

Testing can be an integral part of a developer's work and help them along the way. It can give you assurances that no other understanding of your code can give you. It can save you time, and you don't even need to hate yourself for it. Let's see how.

## 4.1 Types of tests

Software testing is about increasing the confidence in the behavior of software. This is important: tests never guarantee a behavior, but they increase its likelihood quite a lot, as in orders of magnitude. There are many ways to categorize different types of testing, but the most important distinction is how we run or implement it as it affects our time economy the most.

### 4.1.1 Manual testing

Testing can be a manual activity and it usually is for a developer. Developers test their code by running it and inspecting its behavior. Manual tests have their own types too, like end-to-end testing, which means testing every supported scenario on a software from beginning to end. The value that end-to-end testing provides is enormous, but it's time-consuming.

*Code reviews* can be considered a way of testing, albeit weak. You can understand what the code does, and what it will do when run, to a certain extent. You can vaguely see how it fulfills the requirements, but you can't tell for sure. Tests, based on their types, can provide different levels of assurances about how the code will work. In that sense, code reviews can be considered a type of test.

#### What's a code review?

The main purpose of a code review was to examine a code before it gets pushed to the repository and find potential bugs in it. You could do it in a physical meeting together or use a website like GitHub. Unfortunately, over the course of years, it has turned into many different things ranging from a rite of passage, which completely destroys the developer's self-esteem, to a pile of a software architect's unwarranted quotes from the articles they read.

The most important part of a code review is that it's the last moment that you can criticize the code without having to fix it yourself. After a piece of code passes the review, it becomes everyone's code. Because you all have approved it. You can always say, "I wish you'd said that in the code review, Mark," whenever someone brings up your terrible  $O(N^2)$  sort code and put your headphones back on. Just kidding. You should feel ashamed for writing an  $O(N^2)$  sort code, especially after reading the book; and you still blame Mark? You should know better. Get along with your colleagues. You'll need them.

Ideally, code reviews are not about code style or formatting, because there are automated tools called either linters or code analysis tools that can check for those issues. It should be mainly about bugs and the technical debt that the code might introduce to other developers.

### 4.1.2 Automated tests

You are a programmer; you have the gift of writing code. That means you can make the computer do things for you and that includes testing. You can write code that tests your code, so you don't have to. Programmers usually focus on creating tooling for only the software they're developing, not the development process itself, but that's equally important.

Automated tests can differ vastly in terms of their scope and, more importantly, how much they increase your confidence in the behavior of the software. The smallest variant of automated tests are *unit tests*. They are also the easiest to write because they test only a single unit of code: a public function. It needs to be public because testing is supposed to test externally visible interfaces rather than internal details of a class. The definition of unit can sometimes change in the literature, be it a class or a module or some other logical arrangement of those, but I find functions as the target units convenient.

The problem with unit tests is that even though they let you to see if units work okay, they can't guarantee if they work okay *together*. So, you have to test if they get along together too. Those tests are called *integration tests*. Automated UI tests are usually integration tests too if they run the production code to build the correct user interface.

### 4.1.3 Living dangerously: Testing in production

I had once bought a poster of a famous meme for one of our developers. It said, "I don't always test code, but when I do, I do it in *production*." I hung it on the wall right behind his monitor so he would always remember not to do it.

**DEFINITION** In software lingo, the term production means a live environment accessed by actual users where any change affects the actual data. Many developers confuse it with their computer. There is "development" for that. "Development" as a name for runtime environment means code running locally on your machine and not affecting any data that harms production. As a precaution to harm production there is sometimes a production-like remote environment that is similar to production. It's sometimes called "staging," which doesn't affect actual data visible to your site's users.

Testing in production, aka live code, is considered a bad practice; no wonder such a poster exists. The reason is that when you find a failure, you might have already lost users or customers by then. More importantly, when you break production, there is a chance that you might break the workflow of all the development team. You can easily understand that it happened by the disappointed looks and raised eyebrows you get if you're in an open office setting, text messages saying "WTF!!!!???", Slack notification numbers increasing like KITT<sup>1</sup>'s speedometer, or the steam coming out of your boss' ears.

Like any bad practice, testing in production isn't always bad either. If the scenario you introduce isn't part of a frequently used, critical code path, you might get away



with testing in production. That's why Facebook had this mantra, "move fast and break things" because they let the developers assess the impact of the change to the business. They later dropped the slogan after 2016 US elections, but it still has some substance. If it's a small break in an infrequently used feature, it might be okay to live through the fallout and fix it as soon as possible.

Even not testing your code can be okay if you think breaking a scenario isn't something your users would abandon the app for. I managed to run one of the most popular websites in Turkey myself with zero automated tests in its first years, with a lot of errors and a lot of downtime, of course, because hello: no automated tests!

#### 4.1.4 Choosing the right testing methodology

You need to be aware of certain factors about a given scenario that you are trying to implement or change in order to decide how you want to test it. Those are mainly risk and cost. It's similar to what we used to calculate in our minds when our parents put us up to a chore.

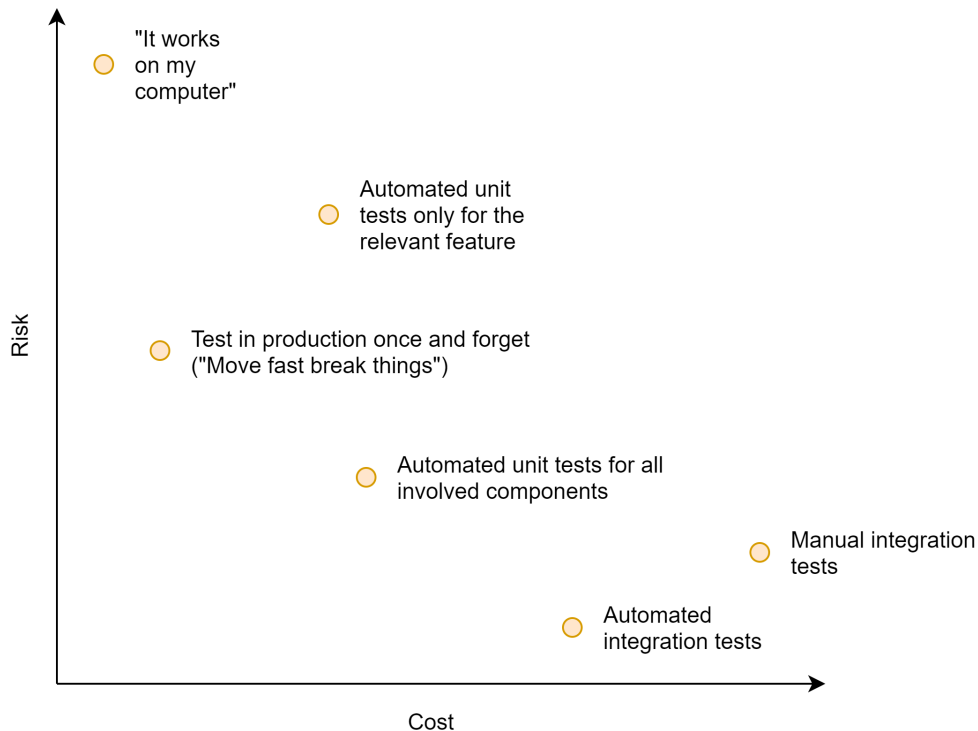
- Cost:
  - How much time do you need to spend to implement/run a certain test?
  - How many times will you need to repeat it?
  - If the code that is tested changes, who will know to test it?
  - How hard is it to keep the test reliable?
- Risk:
  - How likely is this scenario to break?
  - If it breaks, how bad it will impact the business? How much money would you lose, aka "would this get me fired if it breaks"?
  - If it breaks, how many other scenarios will break along with this? For example, if your mailing feature stops working, many features depending on it will be broken too.
  - How frequent does the code change or how much do you anticipate that it will change in the future? Every change introduces a new risk.

You need to find a sweet spot that costs you the least and poses the least risk. Every risk is an implication of more cost to you. In time, you will have a mental tradeoff map for how much cost a test introduces and how much risk it poses, as in figure 4.1.

Never say "it works on my computer" loudly to someone. It's for your internal thinking only. There will never be a code that you can describe as, "Well, it didn't work on my computer, but I was weirdly optimistic!" Of course, it works on your computer! Can you imagine deploying something that you cannot even run yourself? You can use it as a mantra while thinking about whether a feature should be tested or not as long

---

<sup>1</sup> KITT, standing for Knight Industries Two Thousand, is a self-driving car equipped with voice recognition, depicted in the Sci-Fi TV series Knight Rider in the 80's. It's normal that you don't understand this reference as anybody who did is probably dead with the possible exception of David Hasselhoff. That guy is immortal.

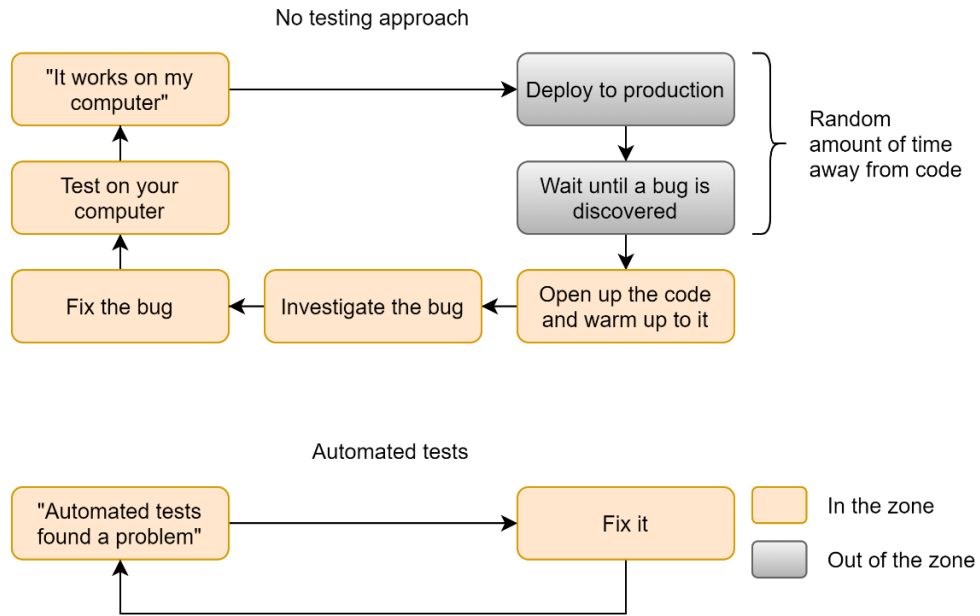


**Figure 4.1** An example mental model to place different options of testing in your mind.

as there is no chain of accountability present. If nobody makes you answer for your mistakes, then go for it. That means the company you're working for has a lot of excess budget to tolerate those mistakes.

If you need to fix your own bugs though, the "It works on my computer" mentality puts you in a very slow and time-wasting cycle due to the delay between the deployment and the feedback loops. One of the basic problems with developer productivity is that interruptions cause significant delays. The reason is *the zone*. We already discussed how warming up to the code can get your productivity wheels turning. That mental state is sometimes called *the zone*. You're in the zone if you're in that productive state of mind. Similarly, getting interrupted can cause those wheels to stop and get you out of the zone, so it forces you to warm up again. As shown in figure 4.2, automated tests alleviate this problem by keeping you in the zone until you reach to a certain degree of confidence about a feature's completion. It shows you two different cycles where how expensive "it works on my computer" can be for both the business and the developer. Every time you get out of the zone you need to spend extra time entering it, which sometimes can even be more than the time you need to test your feature manually.

You can reach a similar quick cycle with manual integration tests too, but they just take more time. That's why automated tests are great: they keep you in the zone and



**Figure 4.2** The expensive development cycle of “it works on my computer” versus “automated tests”.

cost you the least time. Arguably, writing and running tests can be considered as disconnected activities that might push you out of the zone. Yet, running unit tests is extremely fast and supposed to end in seconds. Writing tests is a slightly disconnected activity but it still makes you think about the code you’ve written. You might even consider it as a recap exercise.

This chapter is mostly about unit testing in general because it is in the sweet spot of cost versus risk in figure 4.1.

## 4.2 How to stop worrying and learn to love the tests

Unit testing is about writing test code that tests a single unit of your code, usually a function. You will encounter people arguing this, as what constitutes a unit. Basically, it doesn’t matter much, as long as you can test a given unit in isolation. You can’t test a whole class in a single test anyway. Every test actually tests only a single scenario for a function. So, it’s usual to have multiple tests even for a single function.

Test frameworks make writing tests as easy as possible, but they are not necessary. A test suite can be simply a separate program that runs the tests and shows the results. As a matter of fact, that was the only way to test your program before test frameworks were a thing. I’d like to show you simple code and how unit testing is evolved over time to write tests for a given function as easy as possible.

Let’s consider that you are tasked with changing how the post dates are displayed on a microblogging website called Blabber. The post dates were displayed as a full

date, and according to the new social media fashion, it's more favorable to use acronyms that shows a duration since the post was created in seconds, minutes, hours, and so forth. You need to develop a function that gets a `DateTimeOffset` and convert it into a string that shows a duration since that span of time in text like "3h" for three hours, "2m" for two minutes, or "1s" for one second. It should show only the most significant unit. If the post is three hours two minutes and one second old, it should only show "3h". Listing 4.1 shows such a function.

In listing 4.1, we define an *extension method* to `DateTimeOffset` class in .NET, so we can call it wherever we want like a native method of `DateTimeOffset`.

### Avoid polluting code completion with extension methods

C# provides a nice syntax to define additional methods for a type even if you don't have access to its source. If you prefix the first parameter of a function with `this` keyword, it starts to appear in that type's method list in code completion. It's so convenient that developers like extension methods a lot and tend to make everything an extension method instead of a static method. Say, you have a simple method like this:

```
static class SumHelper {  
    static int Sum(int a, int b) => a + b;  
}
```

In order to call this method, you have to write `SumHelper.Sum(amount, rate)`; and more importantly, you must know that there is a class called `SumHelper`. You can write it as an extension method instead like this:

```
static class SumHelper {  
    static decimal Sum(this int a, int b) => a * b;  
}
```

Now, you can call the method like this:

```
int result = 5.Sum(10);
```

Looks good, but there is a problem. Whenever you write an extension method for a well-known class like `string` or `int`, you introduce it to code completion, which is the drop-down you see on Visual Studio when you type a dot after an identifier. It can be extremely annoying to struggle to find the method you're looking for among the list of completely irrelevant methods.

Do not introduce a purpose-specific method into a commonly used .NET class. Do that only for generic methods that will be used commonly. For example, a `Reverse` method in `String` class can be okay, but `MakeCdnFilename` wouldn't. `Reverse` can be applicable in any context, but `MakeCdnFilename` would only be needed when you must, well, make a filename suitable for the content delivery network you're using. Other than that, it's a nuisance for you and every developer in your team. Don't make people hate you. More importantly, don't make yourself hate you. In those cases, you can perfectly use a static class and a syntax like: `Cdn.MakeFilename()`.

**(continued)**

Don't create an extension method when you can make the method the part of the class. It only makes sense to do that when you want to introduce a new functionality beyond a dependency boundary. For example, you might have a web project that uses a class defined in a library that doesn't depend on web components. Later, you might want to add a specific functionality to that class related to web functionality in the web project. It's preferable to introduce new dependency only to the extension method in the web project, rather than making the library depend on your web components. Unnecessary dependencies can tie your shoelaces together.

We calculate the interval between current time and the post time and check its fields to find out the most significant unit of the interval and return the result based on it.

**Listing 4.1 A function that converts a date to a string representation of the interval**

```
public static class DateTimeExtensions {
    public static string ToIntervalString(
        this DateTimeOffset postTime) {
        TimeSpan interval = DateTimeOffset.Now - postTime;
        if (interval.TotalHours >= 1.0) {
            return $"{(int)interval.TotalHours}h";
        }
        if (interval.TotalMinutes >= 1.0) {
            return $"{(int)interval.TotalMinutes}m";
        }
        if (interval.TotalSeconds >= 1.0) {
            return $"{(int)interval.TotalSeconds}s";
        }
        return "now";
    }
}
```

**this defines an extension method to DateTimeOffset class.**

**Calculate the interval.**

**It's possible to write this code shorter or more performant, but not when it sacrifices readability.**

We have a vague spec about the function, and we can start writing some tests for it. It'd be good idea to write possible inputs and expected outputs in a table to ensure the function works correctly, as in table 4.1.

**Table 4.1 A Sample Test Specification for our Conversion Function**

Input	Output
< 1 second	"now"
< 1 minute	"<seconds>s"
< 1 hour	"<minutes>m"
>= 1 hour	"<hours>h"

If `DateTimeOffset` is a class, we should also be testing for the case when we pass null, but because it's a struct, it cannot be null. That saved us one test. Normally, you don't really need to create a table like that, and you can usually manage a mental model of it, but whenever you're in doubt, feel free to write it down.

Our tests should consist of calls with different `DateTimeOffset`'s and comparisons with different strings. At this point, test reliability becomes a concern because `DateTime.Now` always changes, and our tests are not guaranteed to run in specific time. If there was another test running, or something slows down the computer, you can easily fail the test for the output "now". That means our tests will be flaky and can fail occasionally.

That indicates a problem with our design. A simple solution would be that we could make our function deterministic by passing a `TimeSpan` instead of a `DateTimeOffset` and calculating the difference in the caller instead. As you can see, writing tests around your code helps you identify design problems too, and that's one of the selling points of TDD (Test-Driven Development) approach, which we didn't use here, because we know that we can just go ahead and change the function easily, as in listing 4.2, to receive a `TimeSpan` directly.

#### Listing 4.2 Our refined design

```
public static string ToIntervalString(
    this TimeSpan interval) {
    if (interval.TotalHours >= 1.0) {
        return $"{(int)interval.TotalHours}h";
    }
    if (interval.TotalMinutes >= 1.0) {
        return $"{(int)interval.TotalMinutes}m";
    }
    if (interval.TotalSeconds >= 1.0) {
        return $"{(int)interval.TotalSeconds}s";
    }
    return "now";
}
```

We receive a `TimeSpan` instead.

Our test cases didn't change, but our tests will be much more reliable. More importantly, we decoupled two different tasks, calculating the difference between two dates and converting an interval to a string representation. Deconstructing concerns in code can help you achieve better designs. It can also be a chore to calculate differences and you can have a separate wrapper function for that.

Now, how do we make sure our function works? We can simply push it to production and wait a couple minutes to hear any screams. If not, we're good to go. By the way, is your resumé up to date? No reason, just asking.

We can write a program that tests the function and see the results. An example program would be like the example in listing 4.3. It's a plain console application that references our project and uses `Debug.Assert` method in `System.Diagnostics` namespace to make sure it passes. It ensures that the function returns expected values. Because asserts run only in Debug configuration, we also ensure that the code isn't run in any other configuration at the beginning with a compiler directive.

**Listing 4.3 A primitive unit testing code**

```

#if !DEBUG      ←
#error Debug.Asserts will only run in Debug configuration
#endif
using System;
using System.Diagnostics;
namespace DateUtilsTests {
    public class Program {
        public static void Main(string[] args) {
            var span = TimeSpan.FromSeconds(3);
            Debug.Assert(span.ToIntervalString() == "3s", "3s case failed");
            span = TimeSpan.FromMinutes(5);
            Debug.Assert(span.ToIntervalString() == "5m", "5m case failed");
            span = TimeSpan.FromHours(7);
            Debug.Assert(span.ToIntervalString() == "7h", "7h case failed");
            span = TimeSpan.FromMilliseconds(1);
            Debug.Assert(span.ToIntervalString() == "now", "now case failed");
        }
    }
}

```

**We need the preprocessor statement to make asserts work.**

**Test case for minutes.**

**Test case for seconds.**

**Test case for hours.**

**Test case for less than a second.**

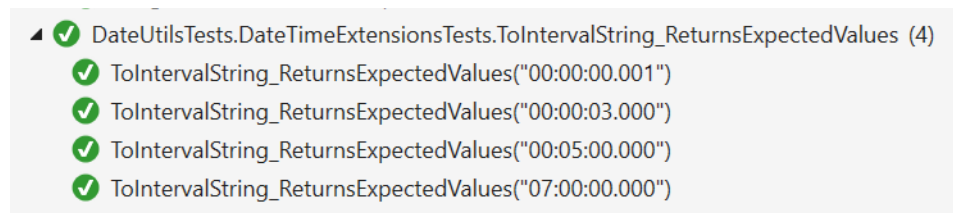
So, why do we need unit test frameworks? Can't we write all tests like this? We could, but it would need more work. In our example, you'll note the following:

- There is no way to detect if any of the tests failed from an external program, such as a build tool. We need special handling around that. Test frameworks and test runners coming with them handle that easily.
- The first failing test would cause the program to terminate. That will lose us time if we have many more failures. We have to run tests again and again, wasting time. Test frameworks can run all tests and report the failures altogether, like compiler errors.
- It's impossible to run certain tests selectively. You might be working on a specific feature and want to debug the function you wrote by debugging the test code. Test frameworks allow you to debug specific tests without having to run the rest.
- Test frameworks can produce code coverage report which helps you identify missing test coverage on your code. It's not possible by writing ad-hoc test code. If you happen to write a coverage analysis tool, you might as well work on creating a test framework.
- Although those tests don't depend on each other, they run sequentially, taking the longest time. Normally, that's not a problem with small number of test cases, but in a medium-scale project, you can have thousands of tests taking different amount of times. You can create threads and run the tests in parallel but that's too much work. Test frameworks can do all of that with a simple switch.
- When an error happens, you only know that there is a problem, but you have no idea about its nature. Strings mismatched, so, what kind of mismatch it is? Did the function return null? Was there an extra character? Test frameworks can report these details too.

- Anything other than using .NET provided `Debug.Assert` will require us writing extra code, a scaffolding if you will. I mean if you start going that path, using an existing framework is much better.
- You'll have the opportunity to join never-ending debates about which test framework is better and feel superior for completely wrong reasons.

Now, let's try writing the same tests with a test framework in listing 4.4. Many test frameworks look alike with the exception of xUnit which is supposedly developed by extraterrestrial life-forms visiting Earth, but in principle it shouldn't matter which framework you're using with the exception of slight changes in the terminology. We're using NUnit here, but you can use any framework you want. You'll see how much clearer the code is with a framework. Most of our test code is actually pretty much a text version of our input/output table in table 4.1. It's apparent what we're testing, and more importantly, although we only have a single test method, we have the capability to run or debug each test individually in the test runner. The technique we used in listing 4.4 with `TestCase` attributes is called *parameterized test*. If you have a specific set of inputs and outputs, you can simply declare them as data and use it in the same function over and over, avoiding repetition over writing a separate test for each test. Similarly, by combining `ExpectedResult` values and declaring the function with a return value, you don't even need to write `Assert`'s explicitly. The framework does it automatically. Less work!

You can run these tests in Test Explorer window of Visual Studio (View → Test Explorer), or you can run `dotnet test` from the command prompt, or you can even use a third-party test runner like NCrunch. The test results in Visual Studio's Test Explorer will look like as in figure 4.3.



**Figure 4.3** Test results that you can't take your eyes off.

#### Listing 4.4 Test framework magic

```
using System;
using NUnit.Framework;
namespace DateUtilsTests {
    class DateUtilsTest {
        [TestCase("00:00:03.000", ExpectedResult = "3s")]
        [TestCase("00:05:00.000", ExpectedResult = "5m")]
        [TestCase("07:00:00.000", ExpectedResult = "7h")]
        [TestCase("00:00:00.001", ExpectedResult = "now")]
    }
}
```



```

public string ToIntervalString_ReturnsExpectedValues(
    string timeSpanText) {
    var input = TimeSpan.Parse(timeSpanText);
    return input.ToIntervalString();
}

```

Converting string to our input type.

No assertions!

You can see how a single function is actually broken into four different functions during test running phase, and how its arguments are displayed along with the test name in figure 4.3. More importantly, you can select a single test, run it, or debug only that test case. And if a test fails you see a brilliant report that exactly tells what's wrong with your code. Say, you accidentally wrote “nov” instead of “now”. The test error would show up like this:

```

Message:
String lengths are both 3. Strings differ at index 2.
Expected: "now"
But was:  "nov"
-----^

```

Not only do you see that there is an error, but you also see a clear explanation about where it happened.

It's a no-brainer to use test frameworks, and you get to love writing tests more when you're aware of how they save you from extra work. They are NASA pre-flight check lights, “system status nominal” announcements, they are your little nanobots doing their work for you. Love tests, love test frameworks.

### 4.3 Don't use TDD or other acronyms

Unit testing, like every successful religion, has split into factions. Test Driven Development (TDD), and Behavior Driven Development (BDD) are some examples. I've come to believe that there are people in software industry who really love to create new paradigms and standards to be followed without question, and there are people who just love to follow without questioning. We love prescriptions and rituals because all you need to do is to follow them, without much thinking. That can cost you a lot of time and make you hate testing too.

The idea behind TDD is that writing tests before actual code can guide you to write better code. TDD prescribes that you should write tests for a class first before writing a single line of code of that class, so the code you write constitutes a guideline on how to implement the actual code. You write your tests. It fails to compile. You start writing actual code, it compiles. Then you run tests and they fail. Then you fix the bugs in your code to make the tests pass. BDD is also a test-first approach with differences in naming and layout of tests.

The philosophy behind TDD/BDD isn't completely rubbish. When you think about how a code should be tested first, it can influence how you think about its design. The problem with TDD isn't the mentality but the practice, the ritualistic approach: write tests and, because the actual code is still missing, get a compiler error (wow, really,

Sherlock?); after writing the code, fix the test failures. I hate errors. They make me feel unsuccessful. Every red squiggly line in the editor, every STOP sign in the Errors list window, every warning icon is a cognitive load, making me confused and distracted.

When you focus on the test before writing a single line of code, you start thinking more about tests than your own problem domain. You start thinking about better ways to write tests. Your mental space gets allocated to the task of writing tests, the test framework's syntactic elements, organization of tests, rather than the production code itself. That's not the goal of testing. Tests shouldn't make you think. Tests should be the easiest piece of code you can write. If that's not the case, you're doing it wrong.

Having tests before writing code triggers the sunk cost fallacy. Remember how dependencies made your code more rigid in chapter 3? Surprise, tests depend on your code too. When you have a full-blown test suite at hand, you become disinclined to change the design of the code because that would mean changing the tests too. It reduces your flexibility when prototyping code. Arguably, tests can give some ideas about if the design really works or not but only in isolated scenarios. You might later discover that a prototype doesn't work well with other components and change your design, before writing any tests. That could be okay if you spend a lot of time on drawing board when designing but that's not usually the case in the streets. You need the ability to quickly change your design.

You can consider writing tests when you believe you're mostly done with your prototype and it seems to be working out okay. Yes, tests will make your code harder to change then, but at the same time it will compensate that by making you confident in the behavior of your code, letting you make changes more easily. You'll effectively get faster.

#### **4.4 Write tests for your own good**

Yes, writing tests improves the software, but it also improves your living standards too. We already discussed how writing tests first can constrain you from changing your code's design. Yet, writing tests last can make your code more flexible because you can easily make significant changes later, without worrying about breaking the behavior after you forget about the code completely. It frees you. It works as insurance, almost the inverse of sunk cost fallacy. The difference of writing tests after is that you are not discouraged in a rapid iteration phase like prototyping. You need to overhaul a code? The first step you need to take is to write tests for it.

Writing tests after you have a good prototype works as a recap exercise for your design. You go over the whole code once again with tests in mind. You can identify certain problems that you didn't find when prototyping your code.

Remember how we discussed doing small, trivial fixes in the code can get you warmed up to large coding tasks? Well, writing tests is one of the great ways to do that. Find missing tests and add them. It never hurts to have more tests as long as they're not redundant. They don't have to be related to your upcoming work. You can simply blindly add test coverage, and who knows, you might find bugs while doing it.

Do you hate your colleagues breaking your code? Tests are there to help. Tests enforce the contract between the code and the specification that developers can't break. You won't have to have comments like this:

```
// When this code was written,  
// only God and I knew what it did.  
// Now only God knows.
```

(That infamous comment block is a derivative joke originally attributed to the author John Paul Friedrich Richter who lived in the 19th century. He didn't write a single line of code, only comments. <https://quoteinvestigator.com/2013/09/24/god-knows/>)

Tests assure you that a fixed bug will remain fixed, and it won't appear again. Every time you fix a bug, adding a test for it, will make sure you won't have to deal with that bug again, ever. Otherwise, who knows when another change won't trigger it again? Tests are critical timesavers when used like that.

Tests improve both the software and the developer. Write tests to be a more efficient developer.

## 4.5 Deciding what to test

*"That is not halted which can eternal run,  
And with strange eons, even tests may be down"*

—H.P. Codecraft

Writing one test, and seeing it pass is only the half of the story. It doesn't mean your function works. Will it fail when the code breaks? Do you cover all the possible scenarios? What should you be testing for? If your tests don't help you to find bugs, they are failures already.

One of my managers had this manual technique to ensure that his team wrote reliable tests: He removed random lines of code from the production code and ran tests again. If your tests passed, that meant you failed.

There are better approaches to find out what cases to test. A specification is a great starting point, but you rarely have those in the streets. It might make sense to create a specification yourself, but even if the only thing you have is code, there are some ways to identify what to test.

### 4.5.1 Respect boundaries

You can call a function that receives a simple integer with four billion different values. Does that mean that you have to test if your function works for each one of those? No. Instead, you should try to identify which input values cause the code to diverge into a branch, or cause values to overflow and test values around those.

Consider a function that checks if a birthdate is of legal age for the registration page of your online game. It's trivial for anyone who was born 18 years before (assuming 18 is the legal age for your game): you just subtract the years and check if it's at

least 18. But what if that person has become 18 last week? Are you going to deprive that person of enjoyment of your pay-to-win game with mediocre graphics? Of course not.

Let's define a function `IsLegalBirthdate`. We use a `DateTime` class instead of `DateTimeOffset` to represent a birthdate because birthdates don't have time zones. If you were born on December 21st in Samoa, your birthday is December 21st everywhere in the world, even in American Samoa, which is 24 hours ahead of Samoa despite being only a hundred miles away. I'm sure there is intense discussion every year about when to have relatives over for Christmas dinner. Time zones are weird.

Anyway, we first calculate the year difference. The only time we need to look at exact dates is for the year of that person's 18th birthday. If it's that year, we check the month and the day. Otherwise, we only check whether the person is older than eighteen. We use a constant to signify legal age instead of writing the number everywhere. Because writing the number is susceptible to typos, and when your boss comes asking you, "Hey can you raise the legal age to 21?" you only have one place to edit it out in this function. You also avoid having to write "// legal age" next to every 18 in the code to explain it. It suddenly becomes self-explanatory. Every conditional in the function—which encompasses if statements, while loops, switch cases, and so forth—causes only certain input values to exercise the code path inside. That means we can split the range of input values based on the conditionals, depending on the input parameters. In our example in listing 4.5, we don't need to test for all possible `DateTime` values between January 1st, 1 and December 31st, 9999, which is about 3.6 million. We only need to test for seven different inputs.

#### Listing 4.5 The bouncer's algorithm

```
public static bool IsLegalBirthdate(DateTime birthdate) {
    const int legalAge = 18;
    var now = DateTime.Now;
    int age = now.Year - birthdate.Year;
    if (age == legalAge) {
        return now.Month > birthdate.Month
            || (now.Month == birthdate.Month
                && now.Day > birthdate.Day);
    }
    return age > legalAge;
}
```

Using conditionals  
in the code.

The seven input values are listed in table 4.2.

**Table 4.2** Partitioning Input Values Based on Conditionals

#	Year difference	Month of birthdate	Day of birthdate	Expected result
1	= 18	= Current month	< Current day	true
2	= 18	= Current month	= Current day	false
3	= 18	= Current month	> Current day	false

**Table 4.2 Partitioning Input Values Based on Conditionals** (continued)

#	Year difference	Month of birthdate	Day of birthdate	Expected result
4	= 18	< Current month	Any	true
5	= 18	> Current month	Any	false
6	> 18	Any	Any	true
7	< 18	Any	Any	false

We suddenly brought down our number of cases from 3.6 million to 7, simply by identifying conditionals. Those conditionals that split the input range are called *boundary conditionals* because they define the boundaries for input values for possible code paths in the function. So, we can go ahead and write tests for those input values as shown in listing 4.6. We basically create a clone of our test table in our inputs and convert it to a `DateTime` and run through our function. We can't hardcode `DateTime` values directly into our input/output table because a birthdate's legality changes based on the current time.

We could convert this to a `TimeSpan`-based function as we did before, but legal age isn't based on exact number of days, but an absolute date time instead. This table is also better because it reflects your mental model more accurately. We use -1 for less than, 1 for greater than, 0 for equality, and prepare our actual input values using those values as references.

#### Listing 4.6 Creating our test function from our input/output table

```
[TestCase(18, 0, -1, ExpectedResult = true)]
[TestCase(18, 0, 0, ExpectedResult = false)]
[TestCase(18, 0, 1, ExpectedResult = false)]
[TestCase(18, -1, 0, ExpectedResult = true)]
[TestCase(18, 1, 0, ExpectedResult = false)]
[TestCase(19, 0, 0, ExpectedResult = true)]
[TestCase(17, 0, 0, ExpectedResult = false)]
public bool IsLegalBirthdate_ReturnsExpectedValues(
    int yearDifference, int monthDifference, int dayDifference) {
    var now = DateTime.Now;
    var input = now.AddYears(-yearDifference)
        .AddMonths(monthDifference)
        .AddDays(dayDifference);
    return DateTimeExtensions.IsLegalBirthdate(input);
}
```

**Preparing our actual input here.**

We did it! We narrowed down the number of possible inputs and identified exactly what to test in our function to create a concrete test plan.

Whenever you need to find out what to test in a function, you're supposed to start with a specification. In the streets though, you'll likely figure out that a specification has never existed or was obsoleted a long time ago so, the second-best way would be to start with boundary conditionals. Using parameterized tests also helps us to focus on

what to test rather than writing repetitive test code. It's occasionally inevitable to create a new function for each test, but specifically with data-bound tests like this, parameterized tests save you considerable time.

### 4.5.2 Code coverage

Code coverage is magic, and like magic, it's mostly stories. Code coverage is measured by injecting every line of your code with callbacks to trace how far the code called by a test executes and which parts it misses. That way, you can find out which part of the code isn't exercised and therefore missing tests.

Development environments rarely come with code-coverage measurement tools out of the box. They are either in astronomically priced versions of Visual Studio, or other paid third-party tools like NCrunch, dotCover, and NCover. Codecov (<https://codecov.io>) is a service that can work with your online repository, and they have a free plan. Free code-coverage measurement locally in .NET was possible only with Coverlet library and Code Coverage reporting extensions in Visual Studio Code at the time of drafting this book.

Code-coverage tools tell you which parts of your code ran when you run your tests. That's quite handy to see what kind of test coverage you're missing to exercise all code paths. It's not the only part of the story, and it's certainly not the most effective. You can have 100% code coverage and still have missing test cases. We'll discuss them later in the chapter.

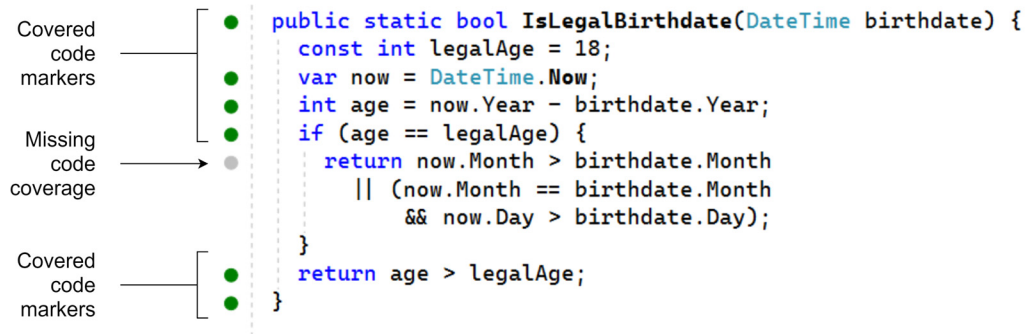
Assume that we comment out the tests where calls to our `IsLegalBirthdate` function with a birthdate that is exactly 18 years old, as in listing 4.7.

#### Listing 4.7 Missing tests

```
// [TestCase(18, 0, -1, ExpectedResult = true)]
// [TestCase(18, 0, 0, ExpectedResult = false)]
// [TestCase(18, 0, 1, ExpectedResult = false)]
// [TestCase(18, -1, 0, ExpectedResult = true)]
// [TestCase(18, 1, 0, ExpectedResult = false)]
[TestCase(19, 0, 0, ExpectedResult = true)]
[TestCase(17, 0, 0, ExpectedResult = false)]
public bool IsLegalBirthdate_ReturnsExpectedValues(
    int yearDifference, int monthDifference, int dayDifference) {
    var now = DateTime.Now;
    var input = now.AddYears(-yearDifference)
        .AddMonths(monthDifference)
        .AddDays(dayDifference);
    return DateTimeExtensions.IsLegalBirthdate(input);
}
```

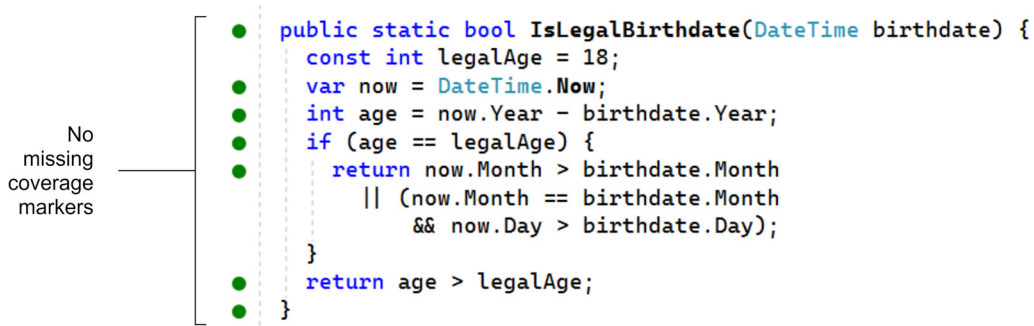
Commented-out test cases.

In that case, a tool like NCrunch, for example, would show the missing coverage as in figure 4.4. The coverage circle next to the return statement inside the if statement is grayed out because we never call the function with a parameter that matches the condition `age == legalAge`. That means we're missing some input values.



**Figure 4.4** Missing code coverage.

When you uncomment those commented out test cases and run tests again, code coverage shows that you have 100% code coverage, as in figure 4.5.



**Figure 4.5** Full code coverage.

Code coverage tools are a good starting point, but they are not fully effective to show you actual test coverage. You should still have a good understanding of the range of input values and boundary conditionals. 100% code coverage doesn't mean 100% test coverage. Consider the following function where you need to return an item from list by index:

```

public Tag GetTagDetails(byte numberOfItems, int index) {
    return GetTrendingTags(numberOfItems)[index];
}

```

Calling that function with `GetTagDetails(1, 0)` would succeed and we would immediately achieve 100% code coverage. Would we have tested all the possible cases? No. Our input coverage would nowhere be close. What if `numberOfItems` is zero and `index` is non-zero? What happens if `index` is negative?

They all mean that we shouldn't be focusing solely on code coverage and trying to fill all the gaps. Instead, we should be conscious about our test coverage by taking all possible inputs into account, being smart about the boundary values. They are not mutually exclusive though; you can use both approaches at the same time.

## 4.6 **Don't write tests**

Yes, testing is helpful, but nothing's better than to avoid writing tests completely. How do you get away without writing tests and still keep your code reliable?

### 4.6.1 **Don't write code**

If a piece of code doesn't exist, it doesn't need to be tested either. There are no bugs in deleted code. Think about this when writing code. Is that something worth writing tests for? Maybe, you don't need to write that code at all. For example, can you opt for using an existing package over implementing it from scratch? Can you leverage an existing class that does the exact same thing you are trying to implement? For example, you might be tempted to write custom regular expressions for validating URL's while all you need to do is to leverage `System.Uri` class.

Third-party code isn't guaranteed to be perfect or always suitable for your purposes of course. You might later discover that the code doesn't work for you. It's usually worth taking that risk before trying to write something from scratch. Similarly, the same codebase you're working on might have the code doing the same job implemented by a colleague. Search your code base to see if something's there.

If nothing works, be ready to implement your own. Don't be scared of reinventing the wheel. It can be very educational, as we discussed in chapter 3.

### 4.6.2 **Don't write all the tests**

The famous *Pareto principle* states that 80% of consequences are the results of 20% of the causes. At least, that's what 80% of the definitions say. It's more commonly called the *80/20 principle*. It's applicable in testing too. You can get 80% reliability from 20% test coverage if you choose your tests wisely.

Bugs don't appear homogenously. Not every code line has the same probability of producing a bug. It's more likely to find bugs in more commonly used code, and code with high churn. You can call those areas of the code *hot paths* where a problem is more likely to happen.

That's exactly what I did with my website. It had no tests whatsoever even after it became one of the most popular Turkish websites in the world. Then, I had to add tests because too many bugs started to appear with the text markup parser. The markup was custom, it barely resembled Markdown, but I developed it before Markdown was even a vitamin in the oranges Dave Gruber ate. Because parsing logic was complicated and prone to bugs, it became economically infeasible to fix every issue after deploying to production. I developed a test suite for it. That was before the advent of test frameworks, so I had to develop my own. I incrementally added more tests as more bugs appeared, because I hated creating the same bugs, and we developed a quite extensive test suite later, which saved us thousands of failing production deployments. Tests just work.

Even just viewing your website's home page provides a good amount of code coverage, because it exercises many shared code paths with other pages. That's called *smoke*



testing around the block. It comes from the times when they developed the first prototype of the computer and just tried to turn it on to see if smoke came out of it. If there was no smoke, that was pretty much a good sign. Similarly, having good test coverage for critical, shared components is more important than having 100% code coverage. Don't spend hours just to add test coverage to that missing line in the constructor of a class if it won't make much difference. You already know that code coverage isn't the whole story.

## 4.7 Let the compiler test your code

With a strongly typed language, you can leverage the type system to reduce the number of test cases needed. We already discussed how nullable references can help you to avoid null checks in the code, which also reduces the need to write tests for null cases. Let's go over a simple example. We already validated that if the person who wants to register is at least 18 years old in the previous section. We now need to validate if the chosen username is valid, so we need a function that validates usernames.

### 4.7.1 Eliminate null checks

Let our rule for a username be lowercase alphanumeric characters up to eight characters long. A regular expression pattern for such a username would be `^[a-z0-9]{1,8}$`. We can write a username class as in listing 4.8. We define a `Username` class to represent all usernames in the code. We avoid the need to think about where we should validate our input by passing this to any code that requires a username.

In order to make sure that a username is never invalid, we validate the parameter in the constructor and throw an exception if it's not in the correct format. Apart from the constructor, the rest of the code is boilerplate to make it work in comparison scenarios. Remember, you can always derive such a class by creating a base `StringValue` class and write minimal code for each string-based value class. I wanted these to remain here to be explicit about what the code entails.

#### Listing 4.8 A username value type implementation

```
public class Username {
    public string Value { get; private set; }
    private const string validUsernamePattern = @"^[a-z0-9]{1,8}$";

    public Username(string username) {
        if (username is null) {
            throw new ArgumentNullException(nameof(username));
        }
        if (!Regex.IsMatch(username, validUsernamePattern)) {
            throw new ArgumentException(nameof(username),
                "Invalid username");
        }
        this.Value = username;
    }
}
```

We validate the username here, once and for all.

```

public override string ToString() => base.ToString();
public override int GetHashCode() => Value.GetHashCode();
public override bool Equals(object obj) {    #B
    return obj is Username other && other.Value == Value;
}
public static implicit operator string(Username username) {
    return username.Value;
}
public static bool operator==(Username a, Username b) {
    return a.Value == b.Value;
}
public static bool operator!=(Username a, Username b) {
    return !(a == b);
}
}

```

**Our usual boilerplate to make a class comparable.**

### Myths around regular expressions

Regular expressions are one of the most brilliant inventions in the history of computer science. We owe them to the venerable Stephen Cole Kleene. They let you create a text parser out of a couple of characters. The pattern "light" matches only the string "light" while "[ln]ight" matches both "light" and "night". Similarly, "li(gh){1,2}t" matches only the words "light" and "lighght" which is not a typo but a single-word Aram Saroyan poem.

Jamie Zawinski famously said, "Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems." The phrase "regular expression" implies certain parsing characteristics. Regular expressions are not context aware, therefore you can't use a single regular expression to find innermost tag in an HTML document or detect unmatched closing tags. That means they are not suitable for complicated parsing tasks. Yet, you can use them to parse text with a non-nested structure.

Regular expressions are surprisingly performant for the cases they are suitable for. If you need extra performance, you can pre-compile them in C# by creating a `Regex` object with the option `RegexOptions.Compiled`. That means a custom code that parses a string based on your pattern will be created on demand. Your pattern turns into C# and eventually machine code. Consecutive calls to the same `Regex` object will reuse the compiled code, gaining you performance for multiple iterations.

Despite how performant they are, you shouldn't use regular expressions when a simpler alternative exists. If you need to check if a string is a certain length, a simple `str.Length == 5` would be way faster and more readable than `Regex.IsMatch(@"^.{5}$", str)`. Similarly, the string class contains many performant methods for common string check operations like `StartsWith`, `EndsWith`, `IndexOf`, `LastIndexOf`, `IsNullOrEmpty`, and `IsNullOrWhiteSpace`. Always prefer them over regular expressions for their specific use cases.

That said, it's also important for you to know at least basic syntax of regular expressions because they can be powerful in a development environment too. You can manipulate code in quite complicated ways that can save you from hours of work and all popular text editors support regular expressions for find and replace operations. I'm talking about operations like "I want to move hundreds of bracket characters in the code to the next line only when they appear next to a code line." You can think about correct regular expression pattern for a couple of minutes as opposed to doing it manually for an hour.

Testing the constructor of Username would require us to create three different test methods as in listing 4.9. One for nullability because a different exception type is raised, the other one is for non-null but invalid inputs, and finally for the valid inputs because we need to make sure that it recognizes valid inputs as valid too.

#### Listing 4.9 Tests for Username class

```
class UsernameTest {
    [Test]
    public void ctor_nullUsername_ThrowsArgumentNullException() {
        Assert.Throws<ArgumentNullException>(
            () => new Username(null));
    }

    [TestCase("")]
    [TestCase("Upper")]
    [TestCase("toolongusername")]
    [TestCase("root!!")]
    [TestCase("a b")]
    public void ctor_invalidUsername_ThrowsArgumentException(string username) {
        Assert.Throws<ArgumentException>(
            () => new Username(username));
    }

    [TestCase("a")]
    [TestCase("1")]
    [TestCase("hunter2")]
    [TestCase("12345678")]
    [TestCase("abcdefgh")]
    public void ctor_validUsername_DoesNotThrow(string username) {
        Assert.DoesNotThrow(() => new Username(username));
    }
}
```

Had we enabled nullable references for the project Username class was in, we wouldn't need to write tests for the null case at all. The only exception to that would be when writing a public API, which may not run against a nullable-references-aware code. In that case, you'd still need to check against nulls.

Similarly, declaring Username as a `struct` when suitable would make it a value type, which would also remove the requirement for a null check. Using correct types and correct structures for types would help you reducing number of tests. The compiler would ensure the correctness of our code instead.

Using specific types for our purposes reduces the need for tests. When your registration function receives a Username instead of a string, you don't need to check if registration function validates its arguments. Similarly, when your function receives a URL argument as a Uri class, you don't need to check if your function processes the URL correctly anymore.

### 4.7.2 Eliminate range checks

You can use unsigned integer types to reduce the surface area of invalid input space. You can see unsigned versions of primitive integer types in table 4.3. There you can see the varieties of data types with their possible ranges which might be more suitable for your code. It's also important that you keep in mind whether the type is directly compatible with `int` or not as it's the go-to type of .NET for integers. You probably have already seen these types, but you might not have considered that they can save you from writing extra test cases. For example, if your function needs only positive values, then why bother with `int` and checking for negative values and throwing exceptions? Just receive `uint` instead.

**Table 4.3** Alternative Integer Types with Different Value Ranges

Name	Integer type	Value range	Assignable to <code>int</code> without loss?
<code>int</code>	32-bit signed	-2147483648..2147483647	Duh
<code>uint</code>	32-bit unsigned	0..4294967295	No
<code>long</code>	64-bit signed	-9223372036854775808..9223372036854775807	No
<code>ulong</code>	64-bit unsigned	0..18446744073709551615	No
<code>short</code>	16-bit signed	-32768..32767	Yes
<code>ushort</code>	16-bit unsigned	0..65535	Yes
<code>sbyte</code>	8-bit signed	-128..127	Yes
<code>byte</code>	8-bit unsigned	0..255	Yes

When you use an unsigned type, trying to pass a negative constant value to your function will cause a compiler error. Passing a variable with a negative value is possible only with explicit type casting, which makes you think about whether the value you have is really suitable for that function at the call site. It's not the function's responsibility to validate for negative arguments anymore. Assume that a function needs to return trending tags in your microblogging web site up to only specified number of tags. It receives a number of items to retrieve rows of posts as in listing 4.10.

In listing 4.8, we have a `GetTrendingTags` function which returns items by taking the number of items into account. Notice that the input value is a `byte` instead of `int`, because we don't have any use case more than 255 items in trending tag list. That actually immediately eliminates the cases where an input value can be negative or too large. We don't even need to validate the input anymore. One fewer test case and a much better range of input values, which reduces the area for bugs immediately.

**Listing 4.10 Receive posts only belonging a certain page**

```

using System;
using System.Collections.Generic;
using System.Linq;

namespace Posts {
    public class Tag {
        public Guid Id { get; set; }
        public string Title { get; set; }
    }

    public class PostService {
        public const int MaxPageSize = 100;
        private readonly IPostRepository db;

        public PostService(IPostRepository db) {
            this.db = db;
        }

        public IList<Tag> GetTrendingTags(byte numberOfItems) {
            return db.GetTrendingTagTable()
                .Take(numberOfItems)
                .ToList();
        }
    }
}

```

**We chose byte instead of int.**

**A byte or a ushort can be passed as safely as int too.**

There are two things happening here. First, we chose a smaller data type for our use case. We don't intend to support billions of rows in a trending tag box. We don't even know what that would look like. We narrowed down our input space. Second, we chose byte, an unsigned type, that's impossible to become negative. That way, we avoided a possible test case and a potential problem that might cause an exception. LINQ's Take function doesn't throw an exception with a List, but it can when it gets translated to a query for a database like Microsoft SQL Server. By changing the type, we avoided those cases, and we don't need to write tests for them.

Note that .NET uses int as the de facto standard type for many operations like indexing and counting. Opting for a different type might have to be cast and converted into int if you happen to interact with standard .NET components with that type. You need to make sure that you're not digging yourself into a hole. For example, if you need more than 255 items in the future, you'll have to replace all references to bytes with shorts or ints which can be a time-consuming task. You need to make sure that you are saving yourself from writing tests for a worthy cause. You might even find writing additional tests more favorable in many cases rather than dealing with different types. In the end, it's only your comfort and your time that matters despite how powerful it is to use types for hinting at valid value ranges.

### 4.7.3 Eliminate valid value checks

There are times we use values to signify an operation in a function. A very common example is `fopen` function in C programming language. It takes a second string parameter that symbolizes the open mode, which can mean “open for reading”, “open for appending”, “open for writing”, and so forth.

.NET team, decades after C of course, has made a better decision and created separate functions for them. You have `File.Create`, `File.OpenRead`, `File.OpenWrite` methods separately, avoiding the need for an extra parameter and the need for parsing that parameter. It’s impossible to pass along the wrong parameter. It’s impossible for functions to have bugs in parameter parsing because there is no parameter.

It’s common to use such values to signify a type of operation. You should consider separating them into distinct functions instead, which can both convey the intent better, and reduce your test surface.

One of the common ways in C# is to use Boolean parameters to change the logic of the running function. An example would be to have a sorting option in our trending tags retrieval function as in listing 4.11, because assume that we need trending tags in our tag management page too, and it’s better to show them sorted by title there. In contradiction with laws of thermodynamics, developers tend to constantly lose entropy. They always try to make the change with the least entropy, without thinking that how much burden it will be in the future. The first instinct of a developer can be to add a Boolean parameter and be done with it.

#### Listing 4.11 Boolean parameters

```
public IList<Tag> GetTrendingTags(byte numberOfItems,
    bool sortByTitle) {
    var query = db.GetTrendingTagTable();
    if (sortByTitle) {
        query = query.OrderBy(p => p.Title);
    }
    return query.Take(numberOfItems).ToList();
}
```

Shows a newly added parameter.

Shows a newly introduced conditional.

The problem is, if we keep adding Booleans like this, it can get really complicated because of the combinations of those variables. Let’s say another feature required trending tags from yesterday. We add that in too with other parameters in listing 4.12. Now, our function needs to support combinations of `sortByTitle` and `yesterdayTags` too.

#### Listing 4.12 More Boolean parameters

```
public IList<Tag> GetTrendingTags(byte numberOfItems,
    bool sortByTitle, bool yesterdaysTags) {
    var query = yesterdaysTags
        ? db.GetTrendingTagTable()
        : db.GetYesterdaysTrendingTagTable();
    if (sortByTitle) {
```

More parameters!

More conditionals!

```

        query = query.OrderBy(p => p.Title);
    }
    return query.Take(numberOfItems).ToList();
}

```

There is an ongoing trend here. Our function's complexity increases with every Boolean parameter. Although we have three different use cases, we have four flavors of the function. With every added Boolean parameter, we are creating fictional versions of the function that no one will use, yet someone might someday and get into a bind. A better approach to have a separate function for each client, as in listing 4.13.

#### Listing 4.13 Separate functions

```

public IList<Tag> GetTrendingTags(byte numberOfItems) {
    return db.GetTrendingTagTable()
        .Take(numberOfItems)
        .ToList();
}

public IList<Tag> GetTrendingTagsByTitle(
    byte numberOfItems) {
    return db.GetTrendingTagTable()
        .OrderBy(p => p.Title)
        .Take(numberOfItems)
        .ToList();
}

public IList<Tag> GetYesterdaysTrendingTags(byte numberOfItems) {
    return db.GetYesterdaysTrendingTagTable()
        .Take(numberOfItems)
        .ToList();
}

```

**We separate functionality  
by function names instead  
of parameters.**

We now have one less test case and you get slightly increased performance for free as a bonus. The gains are miniscule of course, and unnoticeable for a single function, but at points where the code needs to scale, they can make a difference without you even knowing. The savings will increase exponentially when you avoid trying to pass state in parameters and leverage functions as much as possible. You might still be irked by repetitive code, which can easily be refactored into common functions as in listing 4.14.

#### Listing 4.14 Separate functions with common logic refactored out

```

private IList<Tag> toListTrimmed(byte numberOfItems,
    IQueryable<Tag> query) {
    return query.Take(numberOfItems).ToList();
}

public IList<Tag> GetTrendingTags(byte numberOfItems) {
    return toListTrimmed(numberOfItems, db.GetTrendingTagTable());
}

public IList<Tag> GetTrendingTagsByTitle(byte numberOfItems) {

```

**Shows common  
functionality.**

```

        return toListTrimmed(numberOfItems, db.GetTrendingTagTable()
            .OrderBy(p => p.Title));
    }

    public IList<Tag> GetYesterdaysTrendingTags(byte numberOfItems) {
        return toListTrimmed(numberOfItems,
            db.GetYesterdaysTrendingTagTable());
    }

```

Our savings are not impressive here, but such refactors can make greater differences in other cases. The important takeaway is to use refactoring to avoid code repetition and combinatorial hell.

The same technique can be used with enum parameters that are used to dictate a certain operation to a function. Use separate functions, and you can even use function composition, instead of passing along a shopping list of parameters.

## 4.8 **Naming tests**

There is a lot in a name. That’s why it’s important to have good coding conventions in both production and test code, although they shouldn’t necessarily overlap. Tests with good coverage can serve as specifications if they’re named correctly. From the name of a test, you should be able to tell:

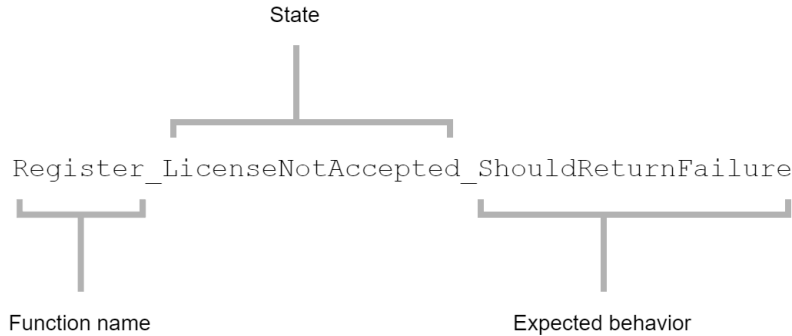
- Name of the function being tested
- Input and initial state
- Expected behavior
- Whom to blame

Kidding about the last one, of course. Remember? You already greenlit that code in the code review. You have no right to blame someone else anymore. If anything, you both should be blamed. I commonly use an “A\_B\_C” format to name tests which is quite different than what you’re used to name your regular functions. We used a simpler naming scheme in previous examples because we were able to use the `TestCase` attribute to describe the initial state of the test. I use an additional “\_ReturnsExpectedValues” but you can simply suffix the function name with `Test`. It’s better if you don’t use the function name alone because that might confuse you when it appears in code completion lists. Similarly, if the function doesn’t take any input or doesn’t depend on any initial state, you can skip the part describing that. The purpose here is to make you spend less time dealing with tests, not to put you through a military drill about naming rules.

Say, your boss came and asked you to write a new validation rules for registration form, where you need to make sure registration code returns failure if user hasn’t accepted policy terms. A name for such a test would be `Register_LicenseNotAccepted_ShouldReturnFailure` as in figure 4.6.

That’s not the only naming convention possible. There are people who prefer creating inner classes for each function to be tested and name tests with only state and expected behavior, but I find that unnecessarily cumbersome. It’s important that you pick the convention that works for you the best.





**Figure 4.6** Components of a test name.

## Summary

- It's possible to overcome the disdain for writing tests by not writing many of them in the first place.
- Test-Driven Development and similar paradigms can make you hate writing tests even more. Seek to write tests that spark joy.
- The effort to write tests can be significantly shortened by test frameworks, especially with parameterized, data-driven tests.
- The number of tests cases can be reduced significantly by properly analyzing boundary values of a function input.
- Proper use of types can let you get away from writing many unnecessary tests.
- Tests don't just ensure the quality of the code. They can help you improve your own development skills and throughput too.
- Testing in production can be acceptable as long as your resumé is up to date.

## **A**

---

- AfterAll setup code 38
- AfterEach setup code 38
- alarm fatigue 77
- anonymous function 9
- anonymous function. *See* lambda
- architecture, compilers and 74–75
- arithmetic errors 63
- assertExhausted method 60
- assertion matcher 49–52
  - and example of over-constrained test assertion 50
  - and example of test assertion with poor explainability 50
  - and examples of appropriate 51
- appropriate 51–52
- described 49
- inappropriate 49–51
- automated tests 82
  - and interruptions in developer production 84

## **B**

---

- BeforeAll setup code 38
- BeforeEach setup code 38
- Behavior Driven Development (BDD) 91–92
- behaviors, testing each 23–25
- BigIntegers 64
- boundary conditionals 95
- breakages, unit testing and accurately detecting 20

## **C**

---

- casts, compilers and 70
- code
  - and linking variables 14
  - and sources of confusion 2, 8
  - and working memory overload 12
- annotation 12
- change reversion 11
- complex
  - and lowering cognitive load 7–12
  - reading 3–4
- delocalized 8
- hot paths and 98
- readability 11
- splitting into smaller units 29–32
- superfluous 24
- synonyms 11
- third-party 98
- working memory and reading 5–7

- code coverage 96–97
  - and missing input values 97
  - and missing tests 96
  - example of full 97
- code review 81
- Codecov 96
- code-coverage measurement tools 96
- cognitive compiling 16
- cognitive load 18
  - code refactoring and reducing 7–12
  - code writing 12
  - different types of 5
  - intrinsic 5, 18
- compiler, code testing and 99–106

compilers

- access control and 61
- and definite assignment 61
- and gaining safety 68–70
  - by enforcing capsulation 69
  - by enforcing sequences 68–69
  - by letting compilers detect unused code 69–70
- with definite values 70
- and warnings 77
- and gaining safety
  - by using them as todo lists 68
- described 59
- halting problem 59
- local invariants 75–77
- overview 59
- reachability 60, 78
- type checking 62–63
- using 67–75

conservative analysis 60

cost, choosing methodology for testing and 83

Coverlet library 96

## D

---

deadlock 66

Debug.Assert method 88

defaults 72

dependencies
 

- hard-coded 52–54
- marking 12–14

dependency graph 12, 18

dependency injection 52–55
 

- modularity and 55

dotCover 96

dynamic types 71

## E

---

end-to-end (E2E) tests 55

error scenarios 24–25

exhaustiveness check 60

extension methods 86–87

extraneous cognitive load 6–7
 

- higher 6
- programming and 6

## F

---

functions
 

- private 26
  - made visible, example of 27

- making visible for testing 29
- public 25

fuzz testing 55

## G

---

germane cognitive load 5

golden testing 55

## H

---

halting problem, compilers and 59

## I

---

if statements. *See* ternary operators

infinite loops 64

inherent complexity 5

inheritance 73

integration tests 55, 82

## K

---

Kleene, Stephen Cole 100

## L

---

lambda 9

laziness 72

list comprehensions 10

long-term memory 4

## M

---

manual testing 81

methods, generic 86

micro-architecture 74

multi-threading 65–67

mutation testing 24

## N

---

NCover 96

NCrunch 90, 96

null checks, elimination of 99–101

null dereference 63

nullable reference 99

nullable variables, risk of runtime errors and 63

**O**

out-of-bounds errors 64

**P**

parameterized test 37–38, 90, 95  
     considering pros and cons 38  
 Pareto (80/20) principle 98  
 private helper functions 26  
 production  
     coding conventions and 106  
     testing in 82  
 programming languages  
     halting problem and 60  
     type systems 62  
 programming, as communication 67  
 Python Tutor 16

**R**

race condition 65  
 range checks, elimination of 102–103  
 read-only fields 61  
 refactoring  
     and replacement of unfamiliar language  
         constructs 8–11  
     duplicate code 7  
     inlining code and 8  
     overview 7  
     reverse refactoring 8  
 regression testing 55  
 regular expressions 100  
     when not to use 100  
 return statement 60  
 risk, choosing methodology for testing and 83  
 runtime types 71–72

**S**

shared test setup 38–49, 56  
     and appropriate use of shared  
         configuration 47–49  
     global state 42  
     shared test constants 45–46  
 sharing configuration 39–40, 43–49  
     and defining important configuration within  
         test cases 46  
     example of 44  
     potential problems with 43  
 sharing state 39–43  
     avoiding 42–43

    potential problems with 41  
 short-term memory 4  
 smoke testing 98  
 starvation 67  
 state table 14–18  
     combining with dependency graph 17  
     described 15  
     steps for creating 15–16  
 StringValue class 99  
 sunk cost fallacy 92  
 Sweller, John 5

**T**

ternary operators 10  
 test code, understandable 21  
 Test Driven Development (TDD) 91–92  
 test frameworks  
     and code coverage report 89  
     benefits of 89  
     example of 90  
     overview 85  
 testing  
     agnostic to implementation details 20, 28  
     and choosing methodology for 83–85  
     as an integral part of developer's work 81  
     automated test 82  
     BDD and 91  
     behaviors 21–25  
         double checking 24  
         each in its own test case 35–37  
         multiple 33–35  
         one at a time 32–38  
     boundary conditions 23  
     code review 81  
     compared to coding 80  
     different values 23  
     end-to-end 81  
     extension methods and 86–87  
     in production 82  
     manual 81  
     only functions 23  
     private functions 26–28  
         example of 27  
     reason for dislike 80  
     specification 95  
     TDD and 91  
     via public API 28–29  
     well-explained failures and 20, 36  
 tests  
     analyzing boundaries 93–96  
     and components of test names 106

- and deciding what to test 93–97
- keeping code reliable without writing 98–99
- multiple 85
- naming 106
- test frameworks 85
- types of 81–85
- writing for your own good 92–93

type checker 62

- and different ways of misusing 70–72

## U

---

unchecked exceptions 73

unit test 82

- key features 20–21
- one test per function 21–23

unsigned integer types 102

Username class 99

- tests for 101

## V

---

valid value checks, elimination of 104–106

- Boolean parameters 104
- separate functions 105

## W

---

working memory 3

- code reading and 5–7
- defined 4
- difference between short-term memory and 4
- overview 4
- support for 16

## Z

---

zone

- automated tests and 84
- interruptions in developer productivity and 84