# Five Lines of Code

Christian Clausen

MEAP

MANNING

**MEAP Edition**
**Manning Early Access Program**
# Five Lines of Code
**Version 4**



For more information on this and other Manning titles go to
manning.com

# *welcome*

Thank you for purchasing *Five Lines of Code* in MEAP.

This book is written to be a first look into refactoring, and is therefore suited for junior programmers. To get the most out of this book you should be comfortable with programming in an object-oriented language, including loops, methods, classes, and interfaces.

We all know the excitement of starting a new project, we open an empty file and start typing, and we don't stop. Our fingers are constantly typing because there is no need to slow down. We can keep the whole codebase in our head, no need to stop and investigate what some other code is doing.

The most productive we can be is when there is no other code to distract us. Sadly in many projects this quickly dissipates, as the codebase gets large and tangled. Changing something in one place breaks something in a completely separate place. We lose the overview, we have to be careful, and it starts sucking the fun out of programming.

Refactoring is the process of improving code without affecting what it does. It makes the code more readable, more flexible and more stable. This book teaches you not only how to refactor, but just as importantly what code to refactor to get the biggest benefits fastest.

In the first part we learn the concrete rules and refactoring patterns through refactoring the codebase of a 2d puzzle game written in a realistic style. In the second part we look at real-world practices that enable great refactoring.

If you have any comments, questions, or feedback while reading it, I will be grateful if you post it in the liveBook Discussion forum. This way you can help make this book great!

Thank you very much for your interest in my book.

—Christian Clausen

# brief contents

# 1 Introduction

***This chapter covers:***

- Understanding the elements of refactoring
- Incorporating refactoring into your daily work
- The importance of safety for refactoring
- Introducing the overarching example for part 1

It is well-known that high code quality leads to cheaper maintenance, fewer errors, and happier developers. The most common way to get high code quality is through refactoring. However, the way refactoring is usually taught — with *code smells* and *unit testing* — imposes an unnecessarily high barrier to entry. I believe that anyone can execute simple refactoring patterns safely, with a little practice.

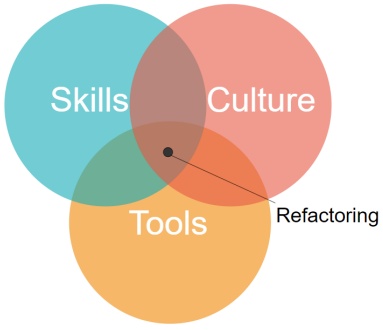


**Figure 1.1. Skills, culture, or tools**

In software development, we place problems somewhere on the diagram shown in figure 1.1, indicating a lack of sufficient skills, culture, tools, or a combination of those. Refactoring is a sophisticated endeavor and therefore lies right in the middle. It requires each component:

- *Skills*. We need the skills to know what code is bad and needs refactoring. Experienced programmers can determine this through their knowledge of code smells. But those are fluffy, or open to interpretation, and not easy to learn; and to a junior developer, understanding code smells can look more like a sixth sense than a skill.
- *Culture*. We need a culture and workflow that encourage taking the time to perform refactoring. In many cases, this is implemented through the famous *red-green-refactor* loop used in test-driven development. However, test-driven development is a much more difficult craft, in my opinion. Red-green-refactor also does not easily give way to doing refactoring in a legacy codebase.
- *Tools*. We need something to help ensure that what we are doing is safe. The most common way to achieve this is through automated testing. But as already mentioned, learning to do effective automated testing is difficult in itself.

The following sections dive into each of these areas and describe how we can begin our refactoring journey from a much simpler foundation, without testing and fluffy code smells smells. Learning refactoring this way can quickly catapult junior developers', students', and programming enthusiasts' code quality to the next level. Tech leads can also use the methods in this book as a basis for introducing refactoring in teams that are not routinely doing it.

## 1.1 What is refactoring?

I answer the question "What is refactoring?" in a lot more detail in the next chapter, but it is helpful to get an intuition for it up front before we dive into the different *how*s of refactoring. In its simplest form, *refactoring* means "changing code without changing what it does." Let's start with an example of refactoring to make it clear what I'm talking about. Here, we replace an expression with a local variable.

**Listing 1.1. Before**

```
return pow(base, exp / 2) * pow(base, exp / 2);
```

**Listing 1.2. After**

```
let result = pow(base, exp / 2);
return result * result;
```

There are many possible reasons to refactor:

- Making code faster (as in the previous example).
- Making code smaller.
- Making code more general or reusable.
- Making code easier to read or maintain.

The last reason is so important and central that we equate it with good code.

**NOTE** **Definition: Good code**

**Good code is human-readable and easy to maintain.**

We discuss these reasons to refactor in more detail in chapter 2. In this book, we only consider refactoring that results in good code; therefore, the definition we use is as follows.

**NOTE** **Our definition: Refactoring**

**Changing code to make it more human-readable and maintainable without changing what it does.**

I should also mention that the type of refactoring we consider relies heavily on working with an object-oriented programming language.

Many people think of programming as writing code; however, most programmers spend more time reading and trying to understand code than writing it. This is because we work in a complex domain, and changing something without understanding it can cause catastrophic failures.

So, the first argument for refactoring is purely economic: programmers' time is expensive, so if we make our codebase more readable, we free up more time for implementing new features. The second argument is that making our code more maintainable means fewer, easier-to-fix bugs. Third, a good codebase is simply more fun. When we read code, we build a model in our heads of what the code is doing; the more we have to keep in our head at one time, the more exhausting it is. This is why it is much more fun to start from scratch — and why debugging can be dreadful.

## 1.2 Skills: What to refactor?

Knowing what you should refactor is the first barrier to entry. Usually, refactoring is taught alongside something called *code smells*. These "smells" are descriptions of things that might suggest our code is bad. While they are powerful, they are also abstract and difficult to get started with, and it takes time to develop a feel for them.

This book takes a different approach and presents easily recognizable, applicable rules to determine what to refactor. These rules are easy to use and quick to learn. They are also sometimes too strict and require you to fix code that is not smelly. On rare occasions, we might follow the rules and still have smelly code.

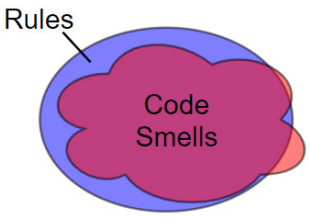


**Figure 1.2. Rules and Code Smells**

As figure 1.2 illustrates the overlap between smells and rules is not perfect. My rules are not the be-all and end-all of good code. They are a head start on the road to developing a guru-like feeling for what good code is.

Let's look at an example of the difference between a code smell and the rules in this book.

### 1.2.1 An Example Code Smell

A well-known code smell is as follows: a function should do *one* thing. This is a great guideline, but it is not easy to know what the *one thing* is. Look again at the earlier code: is it smelly? Arguably, it divides, exponentiates, and then multiplies. Does that mean it does three things? On the other hand, it only returns one number and doesn't change any state, so is it only doing one thing?

```
let result = pow(base, exp / 2);
return result * result;
```

### 1.2.2 An Example Rule

Compare the preceding code smell to the following rule (covered in detail in chapter 3): a method should never have more than *Five Lines of Code*. We can determine this at a glance, with no further questions to ask. The rule is clear, concise, and easy to remember — especially since it is also the title of this book.

Remember, the rules presented in this book are like training wheels. As discussed earlier, they cannot guarantee good code in every situation; and on some occasions, it might be wrong to follow them. However, they are useful if you don't know where to start, and they motivate nice code refactoring.

Note that all the names of the rules are stated in absolute terms, using words like *never*, so they are easy to remember. But the detailed descriptions often specify exceptions: when *not* to apply the rules. The descriptions also state the rules' intention. At the beginning of learning refactoring, we only need to use the absolute names; when those are internalized, we can start learning the exceptions as well, after which we can begin to use the intention — because then we'll be gurus.

## 1.3. Culture: When to refactor?

> *Refactoring is like taking a shower.*
>
> — Kent Beck

Refactoring works best — and costs least — if you do it regularly. So if you can, I recommend that you incorporate it into your daily work. Most of the literature suggests a red-green-refactor workflow; but as mentioned earlier, this ties refactoring to test-driven development — and in this book, we want to separate them and focus specifically on the refactoring part. Therefore, I recommend a more general six-step workflow to solve any programming task, as shown in figure 1.3:

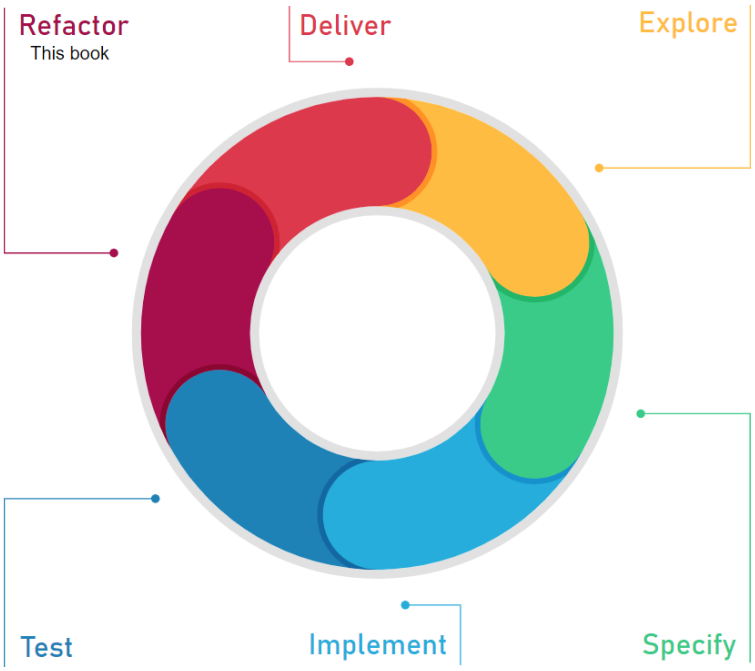


**Figure 1.3. Workflow**

1. *Explore*. Often, we are not completely sure what we need to build right from the start. Sometimes the customer does not know what they want us to build; other times, the requirements are written in ambiguous prose; sometimes we do not even know if the task can be solved. So, always start by experimenting. Implement something quickly, and then you can validate with the customer that you agree on what they need.
2. *Specify*. Once you know what you need to build, make it explicit. Optimally, this results in some form of automated test.
3. *Implement*. Implement the code.
4. *Test*. Make sure the code passes the specification from step 2.
5. *Refactor*. Before delivering the code, make sure it is easy for the next person to work with (and that next person might be you).
6. *Deliver*. There are many ways to deliver; the most common are through a pull request or by pushing to a specific branch. The most important thing is that your code gets to the users. Otherwise, what's the point?

Because we are doing *rule-based* refactoring, the workflow is straightforward and easy to get started with. Figure 1.4 zooms in on step 5: *refactor*.

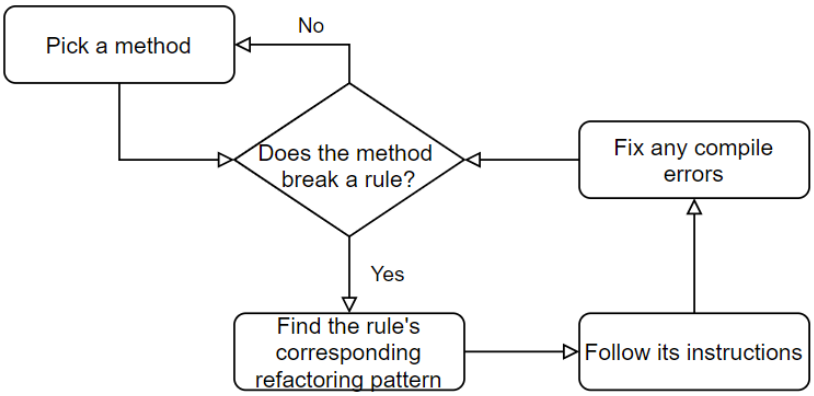


**Figure 1.4. Detailed view of refactoring step**

I have designed the rules so they are easy to remember and so that it's easy to spot when to use them without any assistance. This means that usually, finding a method that breaks a rule is trivial. Every rule also has a few refactoring patterns linked with it, making it easy to know exactly how to fix a problem. The refactoring patterns have explicit step-by-step instructions to ensure that you do not accidentally break something. Many of the refactoring patterns in this book intentionally use compile errors to help make sure you don't introduce errors. Once we've practiced a little, both the rules and the refactoring patterns will become second nature.

### 1.3.1 Introducing refactoring in a legacy system

Even if we are starting from a large legacy system, there is a clever way to incorporate refactoring into our daily work without having to stop everything and refactor the whole codebase first. Simply following this awesome quote:

> *First, make the change easy, then make the easy change.*
>
> — Kent Beck

Whenever we are about to implement something new, we start by refactoring, so it is easy to add our new code. This is similar to getting all the ingredients ready before you start baking.

### 1.3.2 When should you not refactor?

Mostly, refactoring is awesome, but it has a few downsides. Refactoring can be time-consuming, especially if you don't do it regularly. And as mentioned earlier, programmer time is expensive.

There are three types of codebases where refactoring probably isn't worth it:

- Code you are going to write, run only once, and then delete. This is what is known as a *spike* in the *Extreme Programming* community.

- Code that is in maintenance mode before it is going to be retired.
- Code with strict performance requirements, such as an embedded system or a high-end physics engine in a game.

In any other case, I argue that investing in refactoring is the smart choice.

## 1.4 Tools: How to refactor (Safely)?

I like automated tests as much as anybody. However, learning how to test software effectively is a complicated skill in itself. So if you already know how to do automated testing, feel free to use it throughout this book. If you don't, don't worry.

We can think about testing this way: automated testing is to software development what brakes are to cars. Cars don't have brakes because we want to go slowly — they have brakes so we feel safe going fast. The same is true for software: automated tests make us feel safe going fast. In this book, we are learning a completely new skill, so we don't need to go fast.

Instead, I propose relying more heavily on other tools, such as these:

- Detailed, step-by-step, structured refactoring patterns akin to recipes.
- Version control.
- The compiler.

I believe that if the refactoring patterns are carefully designed and performed in tiny steps, it is possible to refactor without breaking anything. This is especially true in cases where our IDE can perform the refactoring for us.

To remedy the fact that we don't talk about testing in this book, we use the compiler and types to catch a lot of the common mistakes we might make. Even so, I recommend that you regularly open the application you are working on and check that it is not completely broken. Whenever we have verified this, or when we know the compiler is happy, we make a commit so that if at some point the application is broken and we don't know how to immediately fix it, we can easily jump back to the last time it was working.

If we are working on a real-world system without automated tests, we can still perform refactoring, but we need to get our confidence from somewhere. Confidence can come from using an IDE to perform the refactoring, testing manually, taking truly tiny steps, or something else. However, the extra time we would spend on these activities probably makes it more cost-effective to do automated testing.

## 1.5 Tools you need to get started

As I said earlier, the type of refactoring discussed in this book need an object-oriented language. That is the primary thing you need to read and understand this book. Coding and refactoring are both crafts that we perform with our fingers. Therefore, they are best learned through the fingers by following along with the examples, experimenting, and having fun while your hands learn the routines. To follow along with the book, you

need the tools described next.

### 1.5.1 Programming language: TypeScript

All the coding examples presented in this book are written in TypeScript.

I chose TypeScript for multiple reasons. Most important, it looks and feels similar to the most commonly used programming languages — Java, C#, C++, and JavaScript — and thus people familiar with any of those languages should be able to read TypeScript without any problem.

TypeScript also provides a way to go from completely "un-object-oriented" code (that is, code without a single class) to highly object-oriented code.

**NOTE** **To better utilize space in the printed book, this book uses a programming style that avoids line breaks while still being readable. I'm not advocating that you use the same style — unless you are coincidentally also writing a book containing lots of TypeScript code.**

If you are unfamiliar with TypeScript, I'll explain any gotchas as they appear, in little boxes like this one:

***In TypeScript...***
We use identity (`===`) to check equality, because it acts more closely to what we expect from equality than double equals (`==`). Consider:

- `0 == ""` is **`true`**.
- `0 === ""` is **`false`**.

Even though the examples are in TypeScript, all refactoring patterns and rules are general and apply to any object-oriented language. In rare cases, TypeScript helps or hinders us; these cases are explicitly stated, and we discuss how to handle these situations in other common languages.

### 1.5.2 Editor: Visual Studio Code

I do not assume that you are using a specific editor; however, if you don't have a preference, I recommend Visual Studio Code. It works well with TypeScript. Also, it supports running `tsc -w` in a background terminal that does the compiling so we don't forget to do it.

**IMPORTANT** ***Visual Studio Code* is different from *Visual Studio***

### 1.5.3 Version control: Git

Although you are not required to use version control to follow along with this book, I strongly recommend it, as it makes it much easier to undo something if you get lost in the middle.

***Resetting to reference solution***
At any point, you can jump to the code as it should look at the beginning of a major section with a command like

```
git reset --hard section-2.1
```

*Caution*: you will lose any changes you have made.

## 1.6 Overarching example: a 2D puzzle game

Finally, let's discuss how I am going to teach all these wonderful rules and amazing refactoring patterns. The book is built around a single overarching example: a 2D block-pushing puzzle game, similar to the classic game Boulder Dash (figure 1.5).

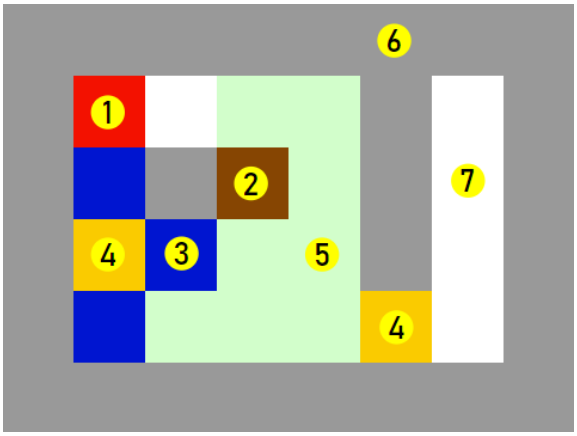


**Figure 1.5. A screenshot of the game out of the box**

This means we have one substantial codebase to play with throughout part 1 of the book. Having one example saves time because we don't have to get familiar with a new example in every chapter.

The example is written in a realistic style, similar to what is used in the industry. It is by no means an easy exercise unless you have the skills learned in this book. The code already adheres to the *DRY*[1] *KISS*[2] principles; even so, it is no more pleasant than a dry kiss.

The reason I chose a computer game is that when we test manually, it is easy to spot if something behaves incorrectly because we have an intuition for how it should behave. It is also slightly more fun to test than looking at something like logs from a financial system.

The user controls the player square using the arrow keys. The objective of the game is to get the box to the lower-right corner:

1. The red square is the player.

2. Brown squares are boxes.
3. Blue squares are stones.
4. Yellow squares are keys *or* locks — we fix this later.
5. Greenish squares are called *flux*.
6. Gray squares are walls.
7. White squares are air (empty).

If a box or stone is not supported by anything, it falls. The player can push one stone or box at a time, provided it is not obstructed or falling. The path between the box and the lower-right corner is initially obstructed by a lock, so the player has to get a key to remove it. Flux can be "eaten" (removed) by the player by stepping on it.

Now would be a great time to get the game and play around with it:

1. Open a console where you want the game to be stored.
   a. `git clone` https://github.com/thedrlambda/five-lines downloads the source code for the game.
   b. `tsc -w` compiles the TypeScript to JavaScript every time it changes.
2. Open `index.html` in a browser.

It is possible to change the level in the code, so feel free to try easier or more difficult levels if you want to:

1. Open the folder in Visual Studio Code.
2. Select Terminal and then New Terminal.
3. Run the command `tsc -w`.
4. TypeScript is now compiling your changes in the background, and you can close the terminal.
5. Every time you make a change, wait for a second, and then refresh your browser.

This is the same procedure you'll use when coding along with the examples in part 1. Before we get to that, though, we build a more detailed foundation of refactoring in the next chapter.

## 1.7 A note on real-world software

It is important to reiterate that the focus of this book is introducing refactoring. The focus is *not* on providing specific rules that you can apply to production code in all circumstances. The way to use the rules is to first learn their names and follow them. Once this is easy for you, learn the description with its exceptions, and finally use this to build an understanding of the underlying code smell. This journey is illustrated in figure 1.6.

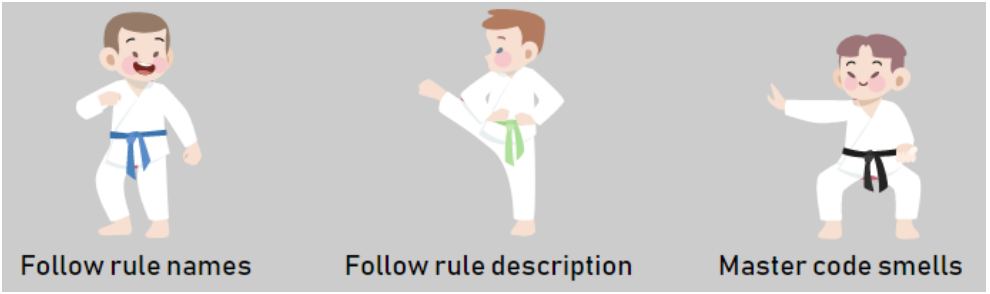


**Figure 1.6. How to use the rules**

This also answers why we cannot make an automatic refactoring program. (We might be able to make a plugin to highlight *possibly problematic* areas in the code, based on the rules.) The purpose of the rules is to build understanding. In short: follow the rules, until you know better.

Also note that because we focus only on learning refactoring, and we have a safe environment, we can get away without automated tests — but this probably is not true for real systems. We do so because it is much easier to learn automated testing and refactoring separately.

## 1.8 Summary

- Executing refactoring requires a combination of *skills* to know what to refactor, *culture* to know when to refactor, and *tools* to know how to refactor.
- Conventionally, code smells are used to describe what to refactor. These are difficult for junior programmers to internalize because they are fuzzy. This book provides concrete rules to replace code smells while learning. The rules have three levels of abstraction: their name is very concrete, and their descriptions add nuance in the form of exceptions and, finally, the intention of the smells they are derived from.
- I believe that automated testing and refactoring can be learned separately, to further lower the barrier to entry. Instead of automated testing, we utilize the compiler, version control, and manual testing.
- The workflow of refactoring is connected with test-driven development in the red-green-refactor loop. But this again implies a dependency on automated testing. Instead, I suggest using a six-step workflow (explore, specify, implement, test, refactor, deliver) for new code or doing refactoring right before changing code.
- Throughout part 1 of this book, we use Visual Studio Code, TypeScript, and Git to transform the source code of a 2D puzzle game.

1. Don't Repeat Yourself.
2. Keep It Simple, Stupid.

# 2 Getting to Know Refactoring; the What, Why, and Why Now?

***This chapter covers:***

- Using readability to communicate intent
- Localizing invariants to improve maintainability
- Enabling change by addition to speed up development
- Making refactoring part of daily work

In the last chapter, we took a look at the different elements involved in refactoring. In this chapter, we dive into the technical details to form a solid foundation of what refactoring is and why it is important from a technical perspective.

## 2.1 Improving readability and maintainability

We start by reiterating the definition of refactoring that we use in this book: refactoring is making code better without changing what it does. Let's break down the two main components of this definition: *making code better* and *without changing what it does*.

### 2.1.1 Making code better

We already saw that better code excels in readability and maintainability and why that matters. But we did not discuss what readability and maintainability are, or how refactoring affects them.

#### READABILITY

*Readability* is the code's aptitude for communicating its intent. It means that if we assume the code works as intended, it is very easy to figure out what it does. There are

several ways to communicate intent in code: having and following conventions; writing comments; variable, method, class, and file naming; using whitespace; and many others.

These techniques can be more or less effective, and we discuss them in detail later. For now, let's look at a simple artificial function that breaks all the communication methods I just described. On the right is the same method without breaking them. One version is hard to read, and the other is easy to read.

**Listing 2.1. Example of really unreadable code**

```
function checkValue(str: boolean) { (1)
  // Check value (2)
  if (str !== false) (3)
    // return (4)
    return true;
  else; // otherwise (5)
    return str; (6)
}
```

(1) Bad method name: a parameter named `str` that is a boolean.
(2) Comment that just repeats a name
(3) Double negation is hard to read.
(4) Comment that just repeats the code
(5) Easy to miss semicolon (`;`) and a trivial comment
(6) Misleading indentation and at this point `str` can only be `false`, so it's clearer to just put that.

**Listing 2.2. Same code written more readably**

```
function isTrue(bool: boolean)
        {
  if (bool)
    return true;
  else
    return false;
}
```

Cleaned up like this, it is clear that we could have simply written the following.

**Listing 2.3. Same code, simplified**

```
function isTrue(bool: boolean) {
  return bool;
}
```

### MAINTAINABILITY

Whenever we need to change some functionality, whether to fix a bug or add a feature, we often start by investigating the context of where we suspect the new code should go. We try to assess what the code is currently doing and how we can safely, quickly, and easily modify it to accommodate our new goal. *Maintainability* is an expression of how much we need to investigate.

It is easy to see that the more code we need to read and include in our investigation, the longer it takes — and the more likely we are to miss something. Therefore, maintainability is closely tied to the risk that is inherent any time we make a change.

Many programmers at every level are deliberate and careful during the investigation phase. Everyone has accidentally missed something at some point and seen the consequences. Being careful also means that if we cannot readily determine whether

something is important, we usually err on the side of caution. Having a long investigation phase is a symptom that code maintainability is bad, and we should strive to improve it.

In some systems, when we change something in one location, something breaks somewhere seemingly unrelated. Imagine an online store where making a change to the recommendation feature breaks the payment subsystem. We call such systems *fragile*.

The root of this fragility is usually *global state*. Here, *global* means outside the scope we are considering. From the perspective of a method, fields are global. The concept of *state* is a bit more abstract; it is anything that can change while our program is running. This includes all the variables, but also the data in a database, files on the hard drive, and the hardware itself. (Technically, even the user's intention and all of reality are state in some sense, but they're unimportant for our purposes.)

A useful trick to help think about global state is to look for braces: `{ … }`. Everything outside the braces is considered global state for everything inside the braces.

The problem with global state is that we often associate properties with our data. The danger is that when data is global, it can be accessed or modified by someone who associates different properties with it, thereby inadvertently breaking our properties. Properties that we do not explicitly check in the code (or check only with assertions) are called *invariants*. "This number will never be negative" and "This file definitely exists" are examples of invariants. Unfortunately, it is nearly impossible to ensure that invariants remain valid, especially as the system changes, programmers forget, and new people are added to the team.

***How Non-Local Invariants Corrupt***

Say we are working on an application for a grocery store. The store sells fruits and vegetables, so in our system, all items have a `daysUntilExpiry` property. We implement a feature that runs every day, subtracts one from `daysUntilExpiry`, and automatically removes items if the value reaches zero. We now have an invariant that `daysUntilExpiry` is always positive.

In our system, we also want an `urgency` property to show how important it is to sell each item. Higher `value` items should have higher `urgency`, and so should items with fewer `daysUntilExpiry`. We therefore implement `urgency = value / daysUntilExpiry`. This cannot go wrong since we *know* that `daysUntilExpiry` is always positive.

Two years later, the store asks us to update the system, because it has started selling light bulbs. We quickly add light bulbs. Light bulbs do not have an expiry date, and we remember the feature that subtracts days and removes items if their `daysUntilExpiry` reaches zero — but we completely forget the invariant. We decide to set `daysUntilExpiry` to zero to start with; this way, it will not be zero after the function subtracts one.

We have violated the invariant, and this results in the system crashing when it tries to calculate the `urgency` of any light bulb: `Error: Division by zero.`

We can improve maintainability by explicitly checking properties, thereby removing invariants. However, this changes what the code does, which refactoring is not allowed to do, as we will see in the next section. Instead, refactoring tends to improve maintainability by moving the invariants closer together, so they are easier to see. This

is called *localizing invariants*: things that change together should be together.

### 2.1.2 Maintaining code ... Without changing what it does

"What does the code do?" is an interesting, albeit somewhat metaphysical, question. Our first instinct is to think of code as a *black box* and say that we may change whatever goes on inside as long as it is indistinguishable from the outside. If we put a value in, we should get the same result before and after a refactoring — even if the result is an exception.

This is mostly true, with one notable exception: we may change performance. Specifically, we rarely care if the code gets slower while refactoring. There are multiple reasons for this. First, in most systems, performance is less valuable than readability and maintainability. Second, if performance is important, it should be handled in a separate phase from refactoring, guided by profiling tools or performance gurus. We discuss optimization in much more detail in a later chapter.

When we refactor, we need to consider the boundaries of our black box. How much code do we intend to change? The more code we include, the more things we can change. This is especially important when working with other people, because if someone makes changes to code we are refactoring, we can end up with nasty merge conflicts. We essentially need to *reserve* the code we are refactoring so no one else changes it. The less code we reserve, the lower the risk of our changes conflicting. As such, determining the appropriate scope of our refactoring is a difficult and important balancing act.

To sum up, the three pillars of refactoring are:

6. Improving readability by communicating intent
7. Improving maintainability by localizing invariants
8. Doing 1 and 2 without affecting any code outside our scope

## 2.2 Gaining speed, flexibility, and stability

I already mentioned the advantages of working in a clean codebase: we are more productive, we make fewer mistakes, and it is more fun. Higher maintainability comes with a few extra perks, which we discuss in this section.

There are several levels of refactoring patterns, from concrete and local (like variable renaming) to abstract and global (like introducing design patterns). While I agree that variable naming can add to or subtract from readability, I believe that the most significant impact on code quality comes from architectural changes. In this book, the closest we come to expression-level refactoring is to discuss good method naming.

### 2.2.1 Composition over inheritance

The fact that non-local invariants are hard to maintain is not new. The endearingly named Gang of Four published the book *Design Patterns* back in 1994, and all those years ago, they recommended against a common way to accidentally introduce non-local invariants: inheritance. Their most famous sentence even tells us how to avoid it:

"Favor object composition over inheritance."

That advice is at the center of this book, and most of the refactoring patterns and rules we describe exist specifically to help with *object composition*: that is, objects having references to other objects. Here is a tiny library for birds (the ornithological details are not important). On the left, it uses inheritance; and on the right, it uses composition.

**Listing 2.4. Using inheritance**

```
interface Bird {
  hasBeak(): boolean;
  canFly(): boolean;
}
class CommonBird implements Bird {
  hasBeak() { return true; }
  canFly() { return true; }
}
class Penguin extends CommonBird { (1)
  canFly() { return false; }
}
```

(1) Inheritance

**Listing 2.5. Using composition**

```
interface Bird {
  hasBeak(): boolean;
  canFly(): boolean;
}
class CommonBird implements Bird {
  hasBeak() { return true; }
  canFly() { return true; }
}
class Penguin implements Bird {
  private bird = new CommonBird(); (2)
  hasBeak() { return bird.hasBeak(); } (3)
  canFly() { return false; }
}
```

(2) Composition

(3) We have to manually forward calls.

In this book, we talk a lot more about the advantages of the right side. But to give a bit of foreshadowing, imagine adding a new method to `Bird` called `canSwim`. In both cases, we add this method to `CommonBird`.

**Listing 2.6. Using inheritance**

```
class CommonBird implements Bird {
  // ...
  canSwim() { return false; }
}
```

In the example with composition, we still have a compiler error in `Penguin`, because it does not implement the new `canSwim` method, so we have to manually add it and decide whether a penguin can swim or not. In the case where we simply want `Penguin` to behave like other birds, this is trivial to implement, like `hasBeak`. Conversely, the inheritance example silently assumes that a `Penguin` cannot swim, so we have to remember to override `canSwim`. Human memory has often proven to be a fragile dependency, especially when our focus is consumed by the new feature we are working on.

### FLEXIBILITY

A system that is built around composition allows us to combine and reuse code in a much more finely grained manner than we could otherwise. Working with systems that use composition heavily is like playing with LEGO blocks. When everything is built to fit together, it is amazingly fast to swap things out or build new things by combining existing components. This flexibility becomes more important when we realize that

most systems end up being used in ways the original programmers didn't imagine.

### 2.2.2 Change by addition

Perhaps the greatest advantage of composition is that it enables *change by addition*. This means it is possible to add or change functionality without affecting other existing functionality — in some cases, without even changing any existing code. We return to how this is technically possible throughout the book; here, we consider some of the implications of change by addition.

This property is also sometimes referred to as the *open-closed principle*, which means components should be open for extension (addition) but closed for modification.

#### PROGRAMMING SPEED

As described earlier, one of the first things we do when we need to implement something new or fix a bug is consider the surrounding code, to ensure that we do not break anything. However, if we can make our changes without touching any of the other code, we can save all that time.

Of course, if we just keep adding code, our codebase quickly grows, which can also be a problem. We need to pay extra attention to what code is being used and what is not. We should delete unused code as quickly as possible. We will also return to this point throughout the book.

#### STABILITY

When we follow a change-by-addition mindset, it is always possible to preserve the existing code. It is easy to implement functionality to fall back on the old functionality if the new code fails. This way, we can ensure that we never introduce new errors in existing functionality. Adding that on top of making fewer errors due to localizing invariants makes for much more stable systems.

## 2.3 Refactoring and your daily work

I said in the introduction that refactoring should be part of any programmer's daily routine. If we deliver unrefactored code, we are only borrowing time from the next programmer. Even worse, due to the negative factors described up to this point, there is an interest rate on this technical debt. I already stated the two variants of daily refactoring that I recommend:

- In a legacy system, start by refactoring before making any changes. Then follow the regular workflow.
- After making any changes to the code, refactor.

Making sure you refactor before you deliver code is also sometimes referred to as

> *Always leave a place better than you found it.*
>
> — The boy scout's rule

### 2.3.1 Refactoring as a method for learning

A final point about refactoring is that, like many things, it takes time to learn; but eventually, it becomes automatic. Seeing and experiencing the advantages of better code changes the way we write and think about code. Once we have a little more stability, we start thinking about how we can exploit this stability. One example of this is increasing our deployment frequency which usually gives even more stability. With flexibility, it is possible to build configuration management or feature toggling systems, the maintenance of which would be unfeasible to without the flexibility.

Refactoring is a completely different way to study code. It gives us a unique perspective. Sometimes we're given code that would take hours or days to understand. The next chapter demonstrates that refactoring allows us to improve code even without understanding it. This way, we can digest small portions while we are working on the code until the final result is very easy to understand.

> ***Refactoring as an intro task***
> Refactoring is often used as an introductory task for new team members, so they can work with the code and learn in a safe environment without having to deal with customers right away. While this is a nice practice, it is only possible if we have neglected our daily due diligence — which I, of course, do not condone.

As I have said, there are many advantages to both learning and practicing refactoring. I hope you are excited to go on this journey with me into the world of refactoring!

## 2.4 Summary

- Refactoring is about making the code communicate its intention and localizing invariants without changing the functionality.
- Favoring composition over inheritance leads to change by addition, by which we gain developer speed, flexibility, and stability.
- We should make refactoring part of our daily work to prevent accumulating technical debt.
- Practicing refactoring gives us a unique perspective on code, which leads us to come up with better solutions.

# 3 Using Functions, to Break Up Long Functions

***This chapter covers:***

- Identifying overly long methods with ***Five lines***
- Working with code without looking at the specifics
- Breaking up long methods with ***Extract method***
- Balancing abstraction levels with ***Either call or pass***
- Isolating `if` statements with ***`if` only at the start***

Code can easily get messy and confusing, even when following the Don't Repeat Yourself and Keep It Simple, Stupid guidelines. Some strong contributors to this messiness are as follows:

- Methods are doing multiple different things.
- We use low-level primitive operations (array accesses, arithmetic operations, etc.).
- We lack human-readable text, like comments and good method and variable naming.

Unfortunately, knowing these issues is not enough to determine exactly what is wrong, let alone how to deal with it.

In this chapter, we describe a concrete way to identify methods that likely have too many responsibilities. As an example, we look at a specific method in our 2D puzzle game that is doing too much: `draw`. We show a structured, safe way to improve the method while eliminating comments. Then, we generalize this process to a reusable refactoring pattern: ***Extract method*** [P3.2.1]. Continuing with the same example `draw` method, we learn how to identify another problem of mixing different

levels of abstraction and how ***Extract method*** [P3.2.1] can also alleviate this issue. In the process, we learn about good method-naming habits.

After concluding our work with `draw`, we continue with another example — the `update` method — and repeat the process, refining how we work with the code without diving into the details of it. This example teaches us to identify a different symptom that a method is doing too much; and through ***Extract method*** [P3.2.1], we learn how to improve readability by renaming variables.

Let's jump into the code, in the file `index.ts`. Remember, you can always check whether your code is up to date with any top-level section in the book by running, for instance, `git diff section-3.1`. If you get lost, you can use, for instance, `git reset --hard section-3.1` to get a clean copy the code at a top-level section. Once we have the code in front of us, we want to improve its quality. But where do we begin?

## 3.1 Our first rule: Why five lines?

To answer this question, we introduce the most fundamental rule in this book: ***Five lines*** [R3.1.1]. This is a simple rule stating that no method should have more than five lines. In this book, ***Five lines*** [R3.1.1] is the ultimate goal, because adhering to this rule is a huge improvement all on its own.

### 3.1.1 *Rule:* Five lines

#### STATEMENT

A method should not contain more than five lines, excluding `{` and `}`.

#### EXPLANATION

A line, sometimes called a *statement*, refers to an **`if`**, a **`for`**, a **`while`**, or anything ending with a semicolon: that is, assignments, method calls, **`return`**, and so on. We discount white space and braces: `{` and `}`.

We can transform any method so it adheres to this rule. Here's an easy way to see how this is possible: if we have a method with 20 lines, we can create a helper method with the first 10 lines and a method with the last 10 lines. The original method is now two lines: one calling the first helper and one calling the second. We can repeat this process until we have as few as two lines in each method.

The specific limit is less important than having a limit. In my experience, it works to set the limit to whatever value is required to implement a pass through your fundamental data structure.

In this book, we are working in a 2D setting, which means our fundamental data structure is a 2D array. The following two functions do a pass through a 2D array: one checks whether the array contains an even number and the other finds the array's minimum element, each in exactly five lines.

**Listing 3.1. Function to check whether a 2D array contains an even number**

```
function containsEven(arr: number[][]) {
  for (let x = 0; x < arr.length; x++) {
    for (let y = 0; y < arr[x].length; y++) {
      if (arr[x][y] % 2 === 0) {
        return true;
      }
    }
  }
  return false;
}
```

***In TypeScript...***

We do not have different types for integers and floating points. We have only one type to cover both: `number`.

**Listing 3.2. Function to find the minimum element in a 2D array**

```
function minimum(arr: number[][]) {
  let result = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++) {
    for (let y = 0; y < arr[x].length; y++) {
      result = Math.min(arr[x][y], result);
    }
  }
  return result;
}
```

***In TypeScript...***

We use **`let`** to declare variables. **`let`** tries to infer the type, but we can specify it with, for example, **`let a: number = 5;`**. We *never* use **`var`**, due to its weird scoping rules: We can define variables after their use. Here, the code on the left is valid, but probably not what we meant. The code on the right gives an error as we expect.

**Listing 3.3. Bad**

```
a = 5;
var a: number;
```

**Listing 3.4. Good**

```
a = 5;
let a: number;
```

To clarify how we count lines, here is the same example we saw at the beginning of chapter 2. We count four lines: one for each **`if`** (including **`else`**) and one for each semicolon.

**Listing 3.5. Four-line method from chapter 2**

```
function isTrue(bool: boolean) {
  if (bool)
    return true;
  else
    return false;
}
```

#### SMELL

Having long methods is a smell in itself. This is because long methods are difficult to work with; you have to keep all the method's logic in your head at once. But "long methods" begs the question: what is *long*?

To answer this question, we draw from another smell: methods should do one thing. If ***Five lines*** is exactly what is necessary to do one meaningful thing, then this limit also prevents us from breaking that smell. We sometimes work in settings where the fundamental data structure is different in different places in the code. Once we are comfortable with this rule, we can start varying the number of lines to fit specific examples. This is fine; but in practice, the number of lines often ends up being around five.

#### INTENT

Left unchecked, methods tend to grow over time as we add more and more functionality to them. This makes them increasingly difficult to understand. Imposing a size limit on our methods prevents us from sliding into this bad territory.

I argue that 4 methods, each with 5 lines of code, can be much more quickly and easily understood than 1 method with 20 lines. This is because each method's name is an opportunity to communicate what the intent of the code is. Essentially, method naming is equivalent to putting a comment at least every five lines. Plus, if small methods are properly named, finding a good name for a big function is easier, too.

#### REFERENCES

To help achieve this rule, see the refactoring ***Extract method*** [P3.2.1]. You can read more about the smell "Methods should do one thing" in Robert C. Martin's book *Clean Code* and the "Long methods" smell in Martin Fowler's book *Refactoring*.

## 3.2 Our first refactoring pattern

While the ***Five lines*** [R3.1.1] rule is easy to understand, achieving it isn't always. Therefore we return to it many times, tackling increasingly difficult examples throughout this part of the book.

With the rule in hand, we are ready to dive into the code. We start with a function named `draw`. Our first stab at understanding the code should always be to consider the function name. The danger is getting bogged down trying to understand every single line – that would take a lot of time and be unproductive. Instead, we begin by looking at the "shape" of the code.

We are trying to identify groups of lines related to the same thing. To make these groups clear, we add blank lines where we think the group should be. Sometimes we add comments to help us remember what the grouping is related to. In general, we strive to avoid comments, as they tend to go out of date, or they are used like deodorant on bad code; but in this case, the comments are temporary, as we'll see in a moment.

Often, the original programmers had groupings in mind and inserted blank lines. Sometimes they included comments. At this point, it is tempting to look at what the code is doing — but since the code is not in a pristine state, that would be counterproductive! You may have heard the saying "The best way to eat an elephant is one bite at a time." This is what we are doing now. Without digesting the entire function, we cut it up and process each piece while it is small and easy to understand.

In figure 3.1, to help avoid getting distracted by the details, we have blurred out all the non-essential lines so we can focus on the structure. (We only do this here in the beginning.) Even without being able to see any specifics, we notice the two groupings, each starting with a comment: `// Draw map` and `// Draw player`.

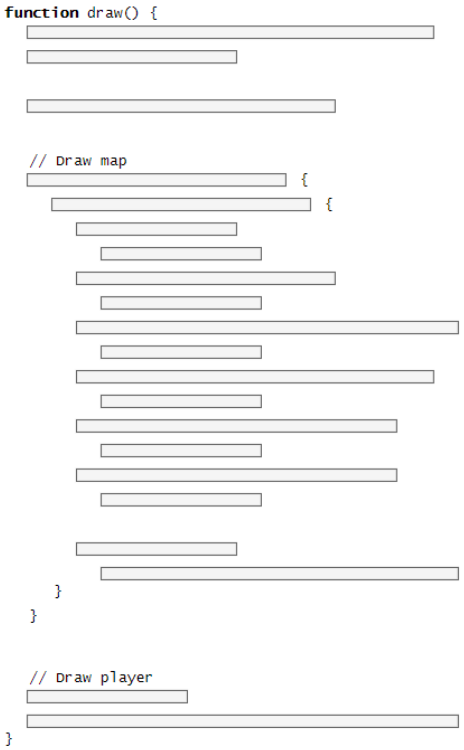


**Figure 3.1. Initial `draw` function**

We can take advantage of those comments by doing the following:

1. Create a new (empty) method, `drawMap`.
2. Where the comment is, put a call to `drawMap`.
3. Select all the lines in the group we identified, and then cut them and paste them as the body of `drawMap`.

Repeating the same process for `drawPlayer` results in the transformation shown in figures 3.2 and 3.3.

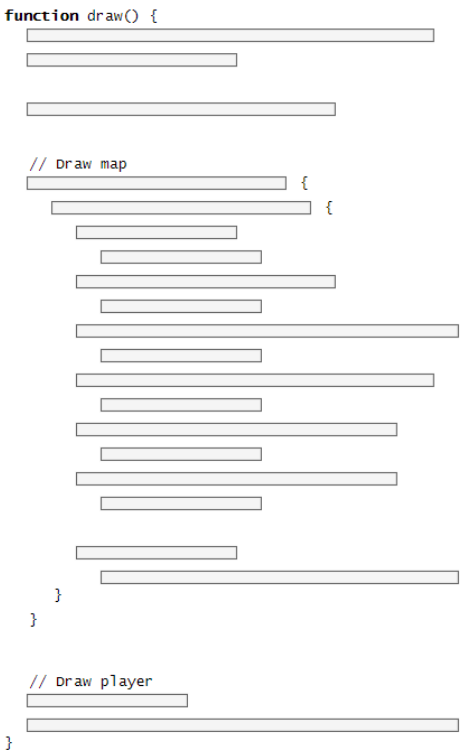


Figure 3.2. Before

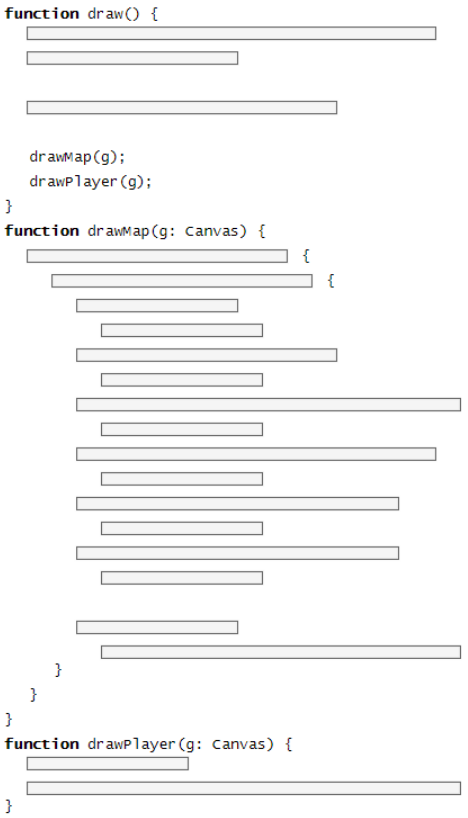


Figure 3.3. After

Now let's take a look at how that works with actual code. We begin with the code in listing 3.6; notice that we can see the same structure, still without looking at what any individual line does.

**Listing 3.6. Initial**

```
function draw() {
  let canvas = document.getElementById("GameCanvas") as HTMLCanvasElement;
  let g = canvas.getContext("2d");

  g.clearRect(0, 0, canvas.width, canvas.height);

  // Draw map (1)
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++) {
      if (map[y][x] === Tile.FLUX)
        g.fillStyle = "#ccffcc";
      else if (map[y][x] === Tile.UNBREAKABLE)
        g.fillStyle = "#999999";
      else if (map[y][x] === Tile.STONE || map[y][x] === Tile.FALLING_STONE)
        g.fillStyle = "#0000cc";
```

```
      else if (map[y][x] === Tile.BOX || map[y][x] === Tile.FALLING_BOX)
        g.fillStyle = "#8b4513";
      else if (map[y][x] === Tile.KEY1 || map[y][x] === Tile.LOCK1)
        g.fillStyle = "#ffcc00";
      else if (map[y][x] === Tile.KEY2 || map[y][x] === Tile.LOCK2)
        g.fillStyle = "#00ccff";

      if (map[y][x] !== Tile.AIR)
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE, TILE_SIZE);
    }
  }

  // Draw player (1)
  g.fillStyle = "#ff0000";
  g.fillRect(playerx * TILE_SIZE, playery * TILE_SIZE, TILE_SIZE, TILE_SIZE);
}
```

(1) Comments marking the start of a logical grouping of lines

***In TypeScript...***

We use **as** to convert between types. It is almost like a cast, except that if the conversion is invalid, it returns `null` instead of crashing.

We follow the steps described earlier:

1. Create a new (empty) method `drawMap`.
2. Where the comment is, put a call to `drawMap`.
3. Select all the lines in the grouping we identified, and then cut them and paste them as the body of `drawMap`.

When we try to compile now, we get quite a few errors. This is because the variable `g` is no longer in scope. We can fix this by first hovering our cursor over `g` in the original `draw` method. This lets us know its type, which we use to introduce a parameter `g: CanvasRenderingContext2D` in `drawMap`.

Compiling again tells us that there is an error where we call `drawMap` because we are missing the parameter `g`. Again, this is easy to fix: we pass `g` as an argument.

Now we repeat the same process for `drawPlayer`, and this is what we end up with — exactly as we expected. Notice that there is still no need to examine what the code is doing any deeper than the method names.

**Listing 3.7. After *Extract method* [P3.2.1]**

```
function draw() {
  let canvas = document.getElementById("GameCanvas") as HTMLCanvasElement;
  let g = canvas.getContext("2d");

  g.clearRect(0, 0, canvas.width, canvas.height);

  drawMap(g); (1)
  drawPlayer(g); (2)
}
```

```
function drawMap(g: CanvasRenderingContext2D) { ❶
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++) {
      if (map[y][x] === Tile.FLUX)
        g.fillStyle = "#ccffcc";
      else if (map[y][x] === Tile.UNBREAKABLE)
        g.fillStyle = "#999999";
      else if (map[y][x] === Tile.STONE || map[y][x] === Tile.FALLING_STONE)
        g.fillStyle = "#0000cc";
      else if (map[y][x] === Tile.BOX || map[y][x] === Tile.FALLING_BOX)
        g.fillStyle = "#8b4513";
      else if (map[y][x] === Tile.KEY1 || map[y][x] === Tile.LOCK1)
        g.fillStyle = "#ffcc00";
      else if (map[y][x] === Tile.KEY2 || map[y][x] === Tile.LOCK2)
        g.fillStyle = "#00ccff";

      if (map[y][x] !== Tile.AIR)
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE, TILE_SIZE);
    }
  }
}

function drawPlayer(g: CanvasRenderingContext2D) { ❷
  g.fillStyle = "#ff0000";
  g.fillRect(playerx * TILE_SIZE, playery * TILE_SIZE, TILE_SIZE, TILE_SIZE);
}
```

❶ New function and call corresponding to the first comment

❷ New function and call corresponding to the second comment

We have completed our first two refactorings: Congratulations! The process we just went through is a standard pattern — a refactoring pattern — that we call ***Extract method*** [P3.2.1].

**NOTE** **Because we are only moving lines around, the risk of introducing errors is minimal, especially since the compiler told us when we forgot parameters.**

We use the comments as the method names; therefore, the functions' names and the comments convey the same information. Thus we eliminate the comments. We also eliminate the now-obsolete blank lines that we used to group the lines.

### 3.2.1 *Refactoring:* Extract method

#### DESCRIPTION

***Extract method*** takes part of one method and extracts it into its own method. This can be done mechanically, and indeed, many modern IDEs have this refactoring pattern built right in. This alone probably makes it safe; computers rarely mess up such things. But there is also a safe way to do it by hand.

Doing so can get complicated if we assign to multiple parameters or **`return`** only in some paths and not all. We do not consider these here as they are rare, and we can

usually simplify them by reordering or duplicating lines in the methods.

### PROCESS

1. Mark the lines to extract by placing blank lines around them, and possibly comments as well.
2. Create a new (empty) method with the desired name.
3. At the top of the grouping, put a call to the new method.
4. Select all the lines in the group, and then cut them and paste them as the body of the new method.
5. Compile.
6. Introduce parameters, thus causing errors.
7. If we assign to *one* of these parameters (let's call it `p`):
   a. Put **`return`** `p;` as the last thing in the new method.
   b. Put the assignment `p = newMethod(…);` at the call site.
8. Compile.
9. Pass arguments, thus causing errors.
10. Remove obsolete blank lines and comments.

### EXAMPLE

Let's see an example of how the full process works. Here we again have a function to find the minimum element in a 2D array. We have determined that it is too long, so we want to extract the part between the blank lines.

**Listing 3.8. Function to find the minimum element of a 2D array**

```
function minimum(arr: number[][]) {
  let result = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++) {
    for (let y = 0; y < arr[x].length; y++) {

      if (result > arr[x][y]) (1)
        result = arr[x][y]; (1)

    }
  }
  return result;
}
```

(1) Lines we want to extract

We follow the process:

1. Mark the lines to extract by placing blank lines around them, and possibly comments as well.
2. Create a new method, `min`.
3. At the top of the grouping, put a call to `min`.
4. Cut and paste the lines in the group into the body of the new method.

**Listing 3.9. Before**

```
function minimum(arr: number[][]) {
  let result =
      Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length;
      x++) {
    for (let y = 0; y <
      arr[x].length; y++) {

      if (result > arr[x][y])
        result = arr[x][y];

    }
  }
  return result;
}
```

**Listing 3.10. After (1/3)**

```
function minimum(arr: number[][]) {
  let result =
      Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length;
      x++) {
    for (let y = 0; y <
      arr[x].length; y++) {

      min(); (1)

    }
  }
  return result;
}

function min() { (1)
  if (result > arr[x][y]) (2)
    result = arr[x][y]; (2)
}
```

(1) New method and call
(2) Extracted lines from before

5. Compile.
6. Introduce parameters for `result`, `arr`, `x`, and `y`.
   a. The extracted function assigns to `result`. So, we need to
   b. Put **`return`** `result;` as the last thing in `min`.
   c. Put the assignment `result = min(…);` at the call site.

**Listing 3.11. Before**

```
function minimum(arr: number[][]) {
  let result = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++) {
    for (let y = 0; y < arr[x].length; y++) {

      min();

    }
  }
  return result;
}

function min() {
  if (result > arr[x][y])
    result = arr[x][y];
}
```

**Listing 3.12. After (2/3)**

```
function minimum(arr: number[][]) {
  let result = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++) {
    for (let y = 0; y < arr[x].length; y++) {

      result = min(); (3)

    }
  }
  return result;
}

function min(
    result: number, (1)
    arr: number[][], (1)
    x: number, (1)
    y: number) { (1)
  if (result > arr[x][y])
    result = arr[x][y];
  return result; (2)
}
```

(1) Added parameters
(2) Added return statement
(3) Assignment to result

7. Compile.
8. We pass the arguments causing errors `result`, `arr`, `x`, and `y`.
9. Finally, we remove the obsolete blank lines.

**Listing 3.13. Before**

```
function minimum(arr: number[][]) {
  let result = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++) {
    for (let y = 0; y < arr[x].length; y++) {

      result = min();

    }
  }
  return result;
}

function min(
    result: number,
    arr: number[][],
    x: number,
    y: number) {
  if (result > arr[x][y])
    result = arr[x][y];
  return result;
}
```

**Listing 3.14. After (3/3)**

```
function minimum(arr: number[][]) {
  let result = Number.POSITIVE_INFINITY;
  for (let x = 0; x < arr.length; x++) {
    for (let y = 0; y < arr[x].length; y++) {
      result =
          min(result, arr, x, y); (1)
    }
  }
  return result;
}

function min(
    result: number,
    arr: number[][],
    x: number,
    y: number) {
  if (result > arr[x][y])
    result = arr[x][y];
  return result;
}
```

(1) Arguments added and blank lines removed

You may be thinking that it would be better to use the build-

in `Math.min` or `arr[x][y]` as an argument instead of all three separately. If you can get there safely, that may be a better approach for you. But the important lesson to take from this example is that the transformation, although slightly cumbersome, is *safe*. We can easily get into trouble trying to be clever, which often isn't worth it.

We can trust that this process does not break anything. The confidence that we have not broken anything is more valuable than perfect output, especially when we have not yet studied what the code does. The more things we have to keep track of, the more likely we are to forget something. The compiler does not forget, and this process is specialized to exploit that fact. We would rather produce unusual-looking code safely than pretty code with less confidence. (If we were feeling confident as a result of something else, like lots of automated testing, we could take more risks; but this isn't the case here.)

#### FURTHER READING

If we want to get a pretty result, we can combine a few other refactoring patterns. We do not go into depth about these, as we only consider statement-level refactoring patterns in this book. But we outline the process here if you want to investigate it further on your own:

1. Execute another small refactoring pattern ,***Extract common subexpression***, which in this case introduces a temporary variable `let tmp = arr[x][y];` outside the grouping and replaces the occurrences of `arr[x][y]` inside the grouping with `tmp`.
2. Use ***Extract method*** as described earlier.
3. Perform ***Inline local variable***, where we undo the work of ***Extract common subexpression*** by replacing `tmp` with `arr[x][y]` and delete the temporary variable `tmp`.

You can read more about all of these patterns, including ***Extract method***, in Martin Fowler's book *Refactoring*.

## 3.3 A second rule

We have achieved the goal of five lines for our seed function, `draw`. Of course, `drawMap` conflicts with the rule; we return to fix this in chapter 4. But we are not quite done with `draw`: it also conflicts with another rule.

### 3.3.1 *Rule:* Either call or pass

#### STATEMENT

A function should either call methods on an object or pass the object as an argument, but not both.

#### EXPLANATION

Once we start introducing more methods and passing things around as parameters, we can end up with uneven responsibilities. For example, a function might be both performing low-level operations, such as setting an index in an array, and also passing

the same array as an argument to a more complicated function. This code would be difficult to read because we would need to switch between low-level operations and high-level method names. It is much easier to stay at one level of abstraction.

Consider this function, which finds the average of an array. Notice that it uses both the high-level abstraction `sum(arr)` and the lowplevel `arr.length`.

**Listing 3.15. Function to find the average of an array**

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

This code violates our rule. Here is a better implementation that abstracts away how to find the length.

**Listing 3.16. Before**

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

**Listing 3.17. After**

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

#### SMELL

The statement "The content of a function should be on the same level of abstraction" is so powerful that it is a smell in its own right. However, as with most other smells, it is hard to quantify what it means, let alone how to address it.

It is trivial to spot whether something is passed as an argument and just as easy to spot if it has a `.` next to it.

#### INTENT

When we introduce abstraction by extracting some details out of a method, this rule forces us to also extract other details. This way, we make sure the level of abstraction inside the method always stays the same.

#### REFERENCES

To help achieve this rule, see the refactoring ***Extract method*** [P3.2.1]. You can read more about the smell "The content of a function should be on the same level of abstraction" in Robert C. Martin's book *Clean Code*.

### 3.3.2 Applying the rule

Again without looking at the specifics, if we examine our `draw` method as it currently looks, in figure 3.4, we quickly spot that we violate this rule. The variable `g` is passed as a parameter, and we also call a method on it.

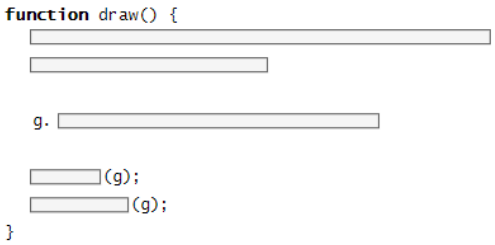


**Figure 3.4. g being both passed and called**

We fix violations of this rule by using ***Extract method*** [P3.2.1]. But what do we extract? Here we need to look a bit at the specifics. There are blank lines in the code, but if we extract the line with `g.clearRect`, we end up passing `canvas` as an argument and also calling `canvas.getContext` — thus violating the rule again.

**Listing 3.18. `draw` as it currently looks**

```
function draw() {
  let canvas = document.getElementById("GameCanvas") as HTMLCanvasElement;
  let g = canvas.getContext("2d");

  g.clearRect(0, 0, canvas.width, canvas.height); (1)

  drawMap(g); (2)
  drawPlayer(g); (2)
}
```

(1) Calls a method on g
(2) g is passed as an argument.

Instead, we decide to extract the first three lines together. Every time we perform ***Extract method*** [P3.2.1], it's a great opportunity to make the code more readable by introducing a good method name.

## 3.4 Naming

I cannot supply universal rules for a *good name*, but I can provide a few properties that a good name should have:

- It should be honest. It should describe the function's intention.
- It should be complete. It should capture everything the function does.
- It should be understandable for someone working in the domain. Use words from the domain you are working in. This also has the advantage of making communication more efficient and making it easier to talk about the code with teammates and customers.

For the first time, we need to consider what the code is doing, because we have no comments to follow. Luckily, we have already significantly reduced the number of lines we need to consider: only three.

The first line fetches the HTML element to draw onto, the second line instantiates the graphics to draw on, and the third clears the canvas. In short, the code creates a graphics object.

**Listing 3.19. Before**

```
function draw() {
  let canvas = document
      .getElementById("GameCanvas")
      as HTMLCanvasElement;
  let g = canvas.getContext("2d");

  g.clearRect(
    0,
    0,
    canvas.width,
    canvas.height);

  drawMap(g);
  drawPlayer(g);
}
```

**Listing 3.20. After**

```
function createGraphics() { (1)
  let canvas = document
      .getElementById("GameCanvas")
      as HTMLCanvasElement; (2)
  let g = canvas.getContext("2d"); (2)
  g.clearRect(
    0,
    0,
    canvas.width,
    canvas.height); (2)
  return g;
}

function draw() {
  let g = createGraphics(); (1)
  drawMap(g);
  drawPlayer(g);
}
```

(1) New method and call

(2) Original lines

Notice that we no longer need any of the blank lines, as the code is easy to understand even without them.

`draw` is finished, and we can move on. Let's start over and go through the same process with another long function: `update`. Again, even without reading any of the code, we can identify two clear groups of lines separated by a blank line.

**Listing 3.21. Initial**

```
function update() {
  while (inputs.length > 0) {
    let current = inputs.pop();
    if (current === Input.LEFT)
      moveHorizontal(-1);
    else if (current === Input.RIGHT)
      moveHorizontal(1);
    else if (current === Input.UP)
      moveVertical(-1);
    else if (current === Input.DOWN)
      moveVertical(1);
  }
  (1)
  for (let y = map.length - 1; y >= 0; y--) {
    for (let x = 0; x < map[y].length; x++) {
      if ((map[y][x] === Tile.STONE || map[y][x] === Tile.FALLING_STONE)
          && map[y + 1][x] === Tile.AIR) {
        map[y + 1][x] = Tile.FALLING_STONE;
        map[y][x] = Tile.AIR;
      } else if ((map[y][x] === Tile.BOX || map[y][x] === Tile.FALLING_BOX)
```

```
          && map[y + 1][x] === Tile.AIR) {
        map[y + 1][x] = Tile.FALLING_BOX;
        map[y][x] = Tile.AIR;
      } else if (map[y][x] === Tile.FALLING_STONE) {
        map[y][x] = Tile.STONE;
      } else if (map[y][x] === Tile.FALLING_BOX) {
        map[y][x] = Tile.BOX;
      }
    }
  }
}
```

❶ Blank line separating two groupings

We can naturally split this code into two smaller functions. What should we call them? Both groups are still pretty complex, so we want to postpone understanding them further. We notice superficially that in the first group, the predominant word is `input`, and in the second, the predominant word is `map`. We know we are splitting a function called `update`, so as a first draft, we can combine these words to get the function names `updateInputs` and `updateMap`. `updateMap` is fine; however, we probably do not "update" the inputs. So, we decide to use another naming trick and use `handle`, instead: `handleInputs`.

**NOTE** **When choosing names like this, always come back later, when the methods are smaller, to assess whether you can improve the names.**

**Listing 3.22. After *Extract method* [P3.2.1]**

```
function update() {
  handleInputs(); ❶
  updateMap(); ❷
}

function handleInputs() { ❶
  while (inputs.length > 0) {
    let current = inputs.pop();
    if (current === Input.LEFT)
      moveHorizontal(-1);
    else if (current === Input.RIGHT)
      moveHorizontal(1);
    else if (current === Input.UP)
      moveVertical(-1);
    else if (current === Input.DOWN)
      moveVertical(1);
  }
}

function updateMap() { ❷
  for (let y = map.length - 1; y >= 0; y--) {
    for (let x = 0; x < map[y].length; x++) {
      if ((map[y][x] === Tile.STONE || map[y][x] === Tile.FALLING_STONE)
          && map[y + 1][x] === Tile.AIR) {
        map[y + 1][x] = Tile.FALLING_STONE;
        map[y][x] = Tile.AIR;
```

```
      } else if ((map[y][x] === Tile.BOX || map[y][x] === Tile.FALLING_BOX)
          && map[y + 1][x] === Tile.AIR) {
        map[y + 1][x] = Tile.FALLING_BOX;
        map[y][x] = Tile.AIR;
      } else if (map[y][x] === Tile.FALLING_STONE) {
        map[y][x] = Tile.STONE;
      } else if (map[y][x] === Tile.FALLING_BOX) {
        map[y][x] = Tile.BOX;
      }
    }
  }
}
```

❶ Extracted first grouping and call

❷ Extracted second grouping and call

Already, `update` is compliant with our rules. We are finished with it. This may not seem like a big deal, but we are getting closer to the magic five lines we are going for.

## 3.5 Refining further

We're finished with `update`, so we can continue with, for instance, one of the functions we just introduced: `updateMap`. In this function, it is not natural to add more white space. Therefore, we need another rule: place ***`if` only at the start*** [R3.5.1] of a function.

### 3.5.1 Rule: `if` only at the start

#### STATEMENT

If you have an **`if`**, it should be the first thing in the function.

#### EXPLANATION

We have already discussed that functions should do only one thing. Checking something is one thing. So, if a function has an **`if`**, it should be the first thing in the function. It should also be the *only* thing, in the sense that we should not do anything after it; but we can avoid having something after it by extracting that separately, as we have seen multiple times.

When we say that **`if`** should be the only thing a method does, we do not need to extract its body, and we also should not separate it from its **`else`**. Both the body and the **`else`** are part of the code structure, and we rely on this structure to guide our efforts so we do not have to understand the code. Behavior and structure are closely tied, and as we are refactoring, we are not supposed to change the behavior — so we shouldn't change the structure, either.

The following example shows a function that prints the primes from 2 to *n*.

**Listing 3.23. Function to print all primes from 2 to *n***

```
function reportPrimes(n: number) {
```

```
  for (let i = 2; i < n; i++)
    if (isPrime(i))
      console.log(`${i} is prime`);
}
```

We have at least two clear responsibilities:

1. Loop over the numbers.
2. Check whether a number is prime.

Therefore, we should have at least two functions.

**Listing 3.24. Before**

```
function reportPrimes(n: number) {
  for (let i = 2; i < n; i++)
    if (isPrime(i))
      console.log(`${i} is prime`);
}
```

**Listing 3.25. After**

```
function reportPrimes(n: number) {
  for (let i = 2; i < n; i++)
    reportIfPrime(i);
}

function reportIfPrime(n: number) {
  if (isPrime(n))
    console.log(`${n} is prime`);
}
```

Every time we check something, it is a responsibility, and it should be handled by one function. Therefore we have this rule.

#### SMELL

This rule — like ***Five lines*** [R3.1.1] — exists to help prevent the smell of functions doing more than one thing.

#### INTENT

This rule intends to isolate **`if`** statements, because they have a single responsibility; and a chain of **`else if`**s represents an atomic unit that we cannot split up. This means the fewest lines we can achieve with ***Extract method*** [P3.2.1] in the context of an **`if`** with **`else if`**s is to extract exactly only that **`if`** along with its **`else if`**s.

#### REFERENCES

To help achieve this rule, see the refactoring ***Extract method*** [P3.2.1]. You can read more about the smell "Methods should do one thing" in Robert C. Martin's book *Clean Code*.

### 3.5.2 Applying the rule

It's easy to spot violations of this rule without looking at the specifics of the code. In figure 3.5, there is one big **`if`** group in the middle of the function.

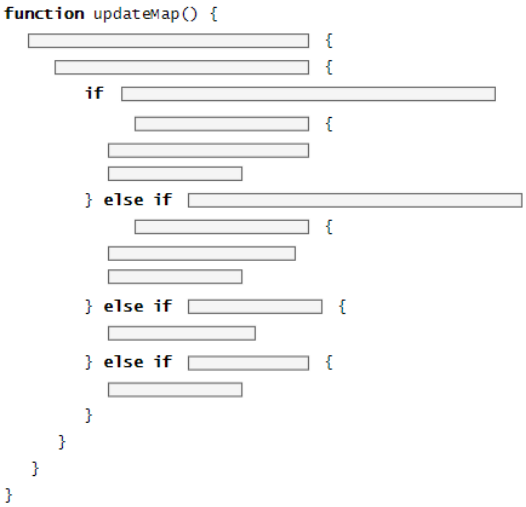

**Figure 3.5.** `if` **in the middle of a function**

To figure out what to name the function that we want to extract, we need to take a superficial look at the code we are extracting. There are two predominant words in this group of lines: `map` and `tile`. We already have `updateMap`, so we call the new function `updateTile`.

**Listing 3.26. After *Extract method* [P3.2.1]**

```
function updateMap() {
  for (let y = map.length - 1; y >= 0; y--) {
    for (let x = 0; x < map[y].length; x++) {
      updateTile(x, y); (1)
    }
  }
}

function updateTile(x: number, y: number) { (1)
  if ((map[y][x] === Tile.STONE || map[y][x] === Tile.FALLING_STONE)
      && map[y + 1][x] === Tile.AIR) {
    map[y + 1][x] = Tile.FALLING_STONE;
    map[y][x] = Tile.AIR;
  } else if ((map[y][x] === Tile.BOX || map[y][x] === Tile.FALLING_BOX)
      && map[y + 1][x] === Tile.AIR) {
    map[y + 1][x] = Tile.FALLING_BOX;
    map[y][x] = Tile.AIR;
  } else if (map[y][x] === Tile.FALLING_STONE) {
    map[y][x] = Tile.STONE;
  } else if (map[y][x] === Tile.FALLING_BOX) {
    map[y][x] = Tile.BOX;
  }
}
```

(1) Extracted method and call

Now `updateMap` is within our five-line limit, and we are content with it. We are starting to feel the momentum, so let's quickly perform the same transformation on `handleInputs`.

**Listing 3.27. Before**

```
function handleInputs() {
  while (inputs.length > 0) {
    let current = inputs.pop();
    if (current === Input.RIGHT)
      moveHorizontal(1);
    else if (current === Input.LEFT)
      moveHorizontal(-1);
    else if (current === Input.DOWN)
      moveVertical(1);
    else if (current === Input.UP)
      moveVertical(-1);
  }
}
```

**Listing 3.28. After**

```
function handleInputs() {
  while (inputs.length > 0) {
    let current = inputs.pop();
    handleInput(current); (1)
  }
}

function handleInput(input: Input) { (1)
  if (input === Input.RIGHT)
    moveHorizontal(1);
  else if (input === Input.LEFT)
    moveHorizontal(-1);
  else if (input === Input.DOWN)
    moveVertical(1);
  else if (input === Input.UP)
    moveVertical(-1);
}
```

(1) Extracted method and call

That completes `handleInputs`. Here we see another readability advantage of ***Extract method*** [P3.2.1]: it lets us give parameters new names that are more informative in their new context. `current` is a fine name for a variable in a loop, but in the new `handleInput` function, `input` is a much better name.

We did introduce a function that seems problematic. `handleInput` is already compact, and it is hard to see how we can make it compliant with the five-line rule. This chapter has only considered ***Extract method*** [P3.2.1] and rules for when to apply it. But since the body of each **`if`** is already a single line, and we cannot extract part of an **`if-else`** chain, we cannot apply ***Extract method*** [P3.2.1] to `handleInput`. However, as we shall see in the next chapter, there is an elegant solution.

## 3.6 Summary

- The ***Five lines*** [R3.1.1] rule states that methods should have five lines or fewer. It helps identify methods that do more than one thing. We use the refactoring pattern ***Extract method*** [P3.2.1] to break up these long methods, and we eliminate comments by making them method names.
- The ***Either call or pass*** [R3.3.1] rule states that a method should either call methods on an object or pass the object as a parameter, but not both. It helps us identify methods that mix multiple levels of abstraction. We again use ***Extract method*** [P3.2.1] to separate different levels of abstraction.
- Method names should be honest, complete, and understandable. ***Extract method*** [P3.2.1] allows us to rename parameters to further improve readability.
- The rule ***`if` only at the start*** [R3.5.1] states that checking a condition using **`if`** does one thing, so a method should not do anything else. This rule also

helps us identify methods that do more than one thing. We use ***Extract method*** [P3.2.1] to isolate these `if`s.

# 4 Using Objects, to Eliminate Complex `if` Statements

***This chapter covers:***

- Eliminating early binding with *Never use* **`if`** *with* **`else`** and *Never use* **`switch`**
- Removing **`if`** statements with *Replace type code with classes* and *Push code into classes*
- Removing bad generalization with *Specialize method*
- Preventing coupling with *Only inherit from interfaces*
- Removing methods with *Inline method* and *Try delete then compile*

At the end of last chapter, we had just introduced a `handleInput` function that we could not use ***Extract method*** [P3.2.1] on because we do not want to break up the **`else if`** chain. Unfortunately, `handleInput` is not compliant with our fundamental ***Five lines*** [R3.1.1] rule, so we cannot leave it as is.

Here's the function:

**Listing 4.1. Initial**

```
function handleInput(input: Input) {
  if (input === Input.LEFT)
    moveHorizontal(-1);
  else if (input === Input.RIGHT)
    moveHorizontal(1);
  else if (input === Input.UP)
    moveVertical(-1);
  else if (input === Input.DOWN)
    moveVertical(1);
}
```

## 4.1 A simple if statement

We are stuck. To show how we deal with **`else if`** chains like this, we start by introducing a new rule.

### 4.1.1 *Rule:* Never use *`if`* with *`else`*

#### STATEMENT

Never use **`if`** with **`else`**, unless we are checking against a data type we do not control.

#### EXPLANATION

Making decisions is hard. In life, many people try to avoid and postpone making decisions; but in code, we seem eager to use **`if`**-**`else`** statements. I won't dictate what is best in real life, but in code, waiting is definitely better. When we put in an **`if`**-**`else`**, we lock in the point at which a decision is made in the code. This makes the code less flexible, as it is not possible to introduce any variation any later than where the **`if`**-**`else`** is located.

We can view **`if`**-**`else`**s as hardcoded decisions. Just as we do not like hardcoded constants in our code, we also do not like hardcoded decisions.

We would prefer never to hardcode a decision — that is, never to use **`if`**s with **`else`**s. Unfortunately, we have to pay attention to what we are checking against. For example, we use `e.keyCode` to check which key is pressed, and it has type `number`. We cannot modify the implementation of `number`, so we cannot avoid an **`else if`** chain.

This should not discourage us, though, because these cases typically occur at the edges of our program, whereas we get input from outside the application: the user typing something, fetching values from a database, and so on. In these cases, the first thing to do is map the third party data types into the data types we have control over. In our example game, one such **`else if`** chain reads the user's input and maps it to our types.

**Listing 4.2. Mapping user input into a data type we control**

```
window.addEventListener("keydown", e => {
  if (e.keyCode === LEFT_KEY || e.key === "a") inputs.push(Input.LEFT);
  else if (e.keyCode === UP_KEY || e.key === "w") inputs.push(Input.UP);
  else if (e.keyCode === RIGHT_KEY || e.key === "d") inputs.push(Input.RIGHT);
  else if (e.keyCode === DOWN_KEY || e.key === "s") inputs.push(Input.DOWN);
});
```

We don't have control over any of the three data types in the conditions: `KeyboardEvent`, `number`, and `string`. As mentioned, these **`else if`** chains should be directly connected to I/O, which should be separated from the rest of the application.

Note that we consider standalone **`if`**s to be *checks*, and **`if`**-**`else`**s to be *decisions*. This allows for simple validation at the start of methods where it would be difficult to

extract an early **`return`**, as in the next example. So, this rule specifically targets **`else`**.

Other from that, this rule is easy to validate: simply look for **`else`**. Let's revisit an earlier function that takes an array of numbers and gives the average. If we call the previous implementation with an empty array, we get a "division by zero" error. This makes sense because we know the implementation, but it is not helpful for the user; so, we would like to throw a more informative error. Here are two ways to fix that.

**Listing 4.3. Bad**

```
function average(ar: number[]) {
  if (length(ar) === 0)
    throw "Empty array not allowed";
  else
    return sum(ar) / length(ar);
}
```

**Listing 4.4. Good**

```
function average(ar: number[]) {
  assertNotEmpty(ar);
  return sum(ar) / length(ar);
}
function assertNotEmpty(ar: number[]) {
  if (length(ar) === 0)
    throw "Empty array not allowed";
}
```

#### SMELL

This rule relates to *early binding*, which is a smell. When we compile our program, a behavior — like **`if-else`** decisions — is resolved and locked into our application, and cannot be modified without recompiling. The opposite of this is *late binding*, where the behavior is determined at the last possible moment when the code is run.

Early binding prevents change by addition because we can only change the **`if`** statement by modifying it. The late-binding property allows us to use change by addition and microservices, which is desirable, as discussed in chapter 2.

#### INTENT

`If`s are control-flow operators. This means they determine what code to run next. However, object-oriented programming has much stronger control-flow operators: objects. If we use an interface with two implementations, then we can determine what code to run based on which class we instantiate. In essence, this rule forces us to look for ways to use objects, which are stronger, more flexible tools.

#### REFERENCES

We discuss late binding in more detail when we look at the ***Replace type code with classes*** [P4.1.3] and ***Introduce strategy-pattern*** [P5.4.2] refactoring patterns.

### 4.1.2 Applying the rule

The first step to get rid of the **`if-else`** in `handleInput` is to replace the `Input` *enum* with an `Input` *interface*. The values are then replaced with classes. Finally — and this is the brilliant part — because the values are now objects, we can move the code inside the **`if`**s to methods in each of the classes. It takes a few sections to get there, so be patient. Let's go through it step by step:

1. Introduce a new interface with the temporary name `Input2`, with methods for the

four values in our enum.

**Listing 4.5. New interface**

```
enum Input {
  RIGHT, LEFT, UP, DOWN
}
interface Input2 {
  isRight(): boolean;
  isLeft(): boolean;
  isUp(): boolean;
  isDown(): boolean;
}
```

2. Create the four classes corresponding to the four enum values. All the methods except the one corresponding to the class should **`return false`**. Note: these methods are temporary, as we will see later.

**Listing 4.6. New classes**

```
class Right implements Input2 {
  isRight() { return true; } (1)
  isLeft() { return false; } (2)
  isUp() { return false; } (2)
  isDown() { return false; } (2)
}
class Left implements Input2 { ... }
class Up implements Input2 { ... }
class Down implements Input2 { ... }
```

(1) `isRight` returns true in the `Right` class.

(2) The other methods return false.

3. Rename the enum to something like `RawInput`. This causes the compiler to report an error in all the places where we use the enum.

**Listing 4.7. Before**

```
enum Input {
  RIGHT, LEFT, UP, DOWN
}
```

**Listing 4.8. After (1/3)**

```
enum RawInput {
  RIGHT, LEFT, UP, DOWN
}
```

4. Change the types from `Input` to `Input2`, and replace the equality checks with the new methods.

**Listing 4.9. Before**

```
function handleInput(input: Input)
      {
  if (input === Input.LEFT)
    moveHorizontal(-1);
  else if (input === Input.RIGHT)
    moveHorizontal(1);
  else if (input === Input.UP)
    moveVertical(-1);
  else if (input === Input.DOWN)
    moveVertical(1);
}
```

**Listing 4.10. After (2/3)**

```
function handleInput(input: Input2) {
       (1)
  if (input.isLeft()) (2)
    moveHorizontal(-1);
  else if (input.isRight()) (2)
    moveHorizontal(1);
  else if (input.isUp()) (2)
    moveVertical(-1);
  else if (input.isDown()) (2)
    moveVertical(1);
}
```

(1) Change type to use the interface

(2) Use the new methods instead of equality checks.

5. Fix the last errors by changing.

**Listing 4.11. Before**

```
Input.RIGHT
Input.LEFT
Input.UP
Input.DOWN
```

**Listing 4.12. After (3/3)**

```
new Right()
new Left()
new Up()
new Down()
```

6. Finally, rename `Input2` to `Input` everywhere.

At this point, here is what the code looks like.

**Listing 4.13. Before**

```
window.addEventListener("keydown", e =>
      {
  if (e.keyCode === LEFT_KEY
        || e.key === "a")
    inputs.push(Input.LEFT);
  else if (e.keyCode === UP_KEY
        || e.key === "w")
    inputs.push(Input.UP);
  else if (e.keyCode === RIGHT_KEY
        || e.key === "d")
    inputs.push(Input.RIGHT);
  else if (e.keyCode === DOWN_KEY
        || e.key === "s")
    inputs.push(Input.DOWN);
});

function handleInput(input: Input) {
  if (input === Input.LEFT)
    moveHorizontal(-1);
  else if (input === Input.RIGHT)
    moveHorizontal(1);
  else if (input === Input.UP)
    moveVertical(-1);
  else if (input === Input.DOWN)
    moveVertical(1);
}
```

**Listing 4.14. After**

```
window.addEventListener("keydown", e => {
  if (e.keyCode === LEFT_KEY
        || e.key === "a")
    inputs.push(new Left());
  else if (e.keyCode === UP_KEY
        || e.key === "w")
    inputs.push(new Up());
  else if (e.keyCode === RIGHT_KEY
        || e.key === "d")
    inputs.push(new Right());
  else if (e.keyCode === DOWN_KEY
        || e.key === "s")
    inputs.push(new Down());
});

function handleInput(input: Input) {
  if (input.isLeft())
    moveHorizontal(-1);
  else if (input.isRight())
    moveHorizontal(1);
  else if (input.isUp())
    moveVertical(-1);
  else if (input.isDown())
    moveVertical(1);
}
```

We capture this process of making enums into classes in the refactoring pattern ***Replace type code with classes*** [P4.1.3].

### 4.1.3 *Refactoring:* Replace type code with classes

#### DESCRIPTION

This refactoring pattern transforms an enum into an interface, and the enums' values become classes. Doing so enables us to add properties to each value and localize functionality concerning that specific value. This leads to change by addition in collaboration with another refactoring pattern, discussed next: ***Push code into classes*** [P4.1.5]. The reason is that we use often use enums via **`switch`**es or **`else if`** chains spread throughout the application. A **`switch`** states how each possible value in an enum should be handled at this location.

When we transform values into classes, we can instead group together functionality concerning that value, without having to consider any other enum values. This process brings functionality and data together, it localizes the functionality to the data, i.e. the specific value. Adding a new value to an enum means verifying logic connected to that enum across many files, whereas adding a new class that implements an interface only asks us to implement methods in that file — no modification of any other code is required (until we want to use the new class).

Note that *type codes* also come in flavors other than enums. Any integer type, or any type that supports the exact equality check `===`, can act as a type code. Most

commonly, we use `int`s and **`enum`**s. Here is an example of such a type code for t-shirt sizes.

**Listing 4.15. Initial**

```
const SMALL = 33;
const MEDIUM = 37;
const LARGE = 42;
```

It is trickier to track down uses of a type code when it is an `int`, because someone might have used the number without reference to a central constant. So we always immediately transform type codes to enums when we see them. Only then can we apply this refactoring pattern safely.

**Listing 4.16. Before**

```
const SMALL = 33;
const MEDIUM = 37;
const LARGE = 42;
```

**Listing 4.17. After**

```
enum TShirtSizes {
  SMALL = 33,
  MEDIUM = 37,
  LARGE = 42
}
```

### PROCESS

1. Introduce a new interface with a temporary name. The interface should contain methods for each of the values in our enum.
2. Create classes corresponding to each of the enum values; all the methods from the interface except the one corresponding to the class should **`return false`**.
3. Rename the enum to something else. Doing so causes the compiler to report an error in all the places where we use the enum.
4. Change types from the old name to the temporary name, and replace equality checks with the new methods.
5. Replace the remaining references to the enum values with instantiating the new classes instead.
6. When there are no more errors, rename the interface to its permanent name everywhere.

### EXAMPLE

Consider this tiny example with a traffic light enum and a function to determine whether we can drive.

**Listing 4.18. Initial**

```
enum TrafficLight {
  RED, YELLOW, GREEN
}
const CYCLE = [TrafficLight.RED, TrafficLight.GREEN, TrafficLight.YELLOW];
function updateCarForLight(current: TrafficLight) {
  if (current === TrafficLight.RED)
    car.stop();
  else
```

```
    car.drive();
}
```

We follow the process:

1. Introduce a new interface with a temporary name. The interface should contain methods for each of the values in our enum.

**Listing 4.19. New interface**

```
interface TrafficLight2 {
  isRed(): boolean;
  isYellow(): boolean;
  isGreen(): boolean;
}
```

2. Create classes corresponding to each of the enum values; all the methods from the interface except the one corresponding to the class should **`return false`**.

**Listing 4.20. New classes**

```
class Red implements TrafficLight2 {
  isRed() { return true; }
  isYellow() { return false; }
  isGreen() { return false; }
}
class Yellow implements TrafficLight2 {
  isRed() { return false; }
  isYellow() { return true; }
  isGreen() { return false; }
}
class Green implements TrafficLight2 {
  isRed() { return false; }
  isYellow() { return false; }
  isGreen() { return true; }
}
```

3. Rename the enum to something else. This causes the compiler to error all the places where we use the enum.

**Listing 4.21. Before**

```
enum TrafficLight {
  RED, YELLOW, GREEN
}
```

**Listing 4.22. After (1/4)**

```
enum RawTrafficLight {
  RED, YELLOW, GREEN
}
```

4. Change types from the old name to the temporary name, and replace equality checks with the new methods.

**Listing 4.23. Before**

```
function updateCarForLight(current:
      TrafficLight) {
  if (current === TrafficLight.RED)
    car.stop();
  else
    car.drive();

}
```

**Listing 4.24. After (2/4)**

```
function updateCarForLight(current:
      TrafficLight2) {
  if (current.isRed())
    car.stop();
  else
    car.drive();
}
```

5. Replace the remaining references to the enum values with instantiating the new classes instead.

**Listing 4.25. Before**

```
const CYCLE = [
  TrafficLight.RED,
  TrafficLight.GREEN,
  TrafficLight.YELLOW
];
```

**Listing 4.26. After (3/4)**

```
const CYCLE = [
  new Red(),
  new Green(),
  new Yellow()
];
```

6. Finally, when there are no more errors, rename the interface to its permanent name everywhere.

**Listing 4.27. Before**

```
interface TrafficLight2 {
  // ...
}
```

**Listing 4.28. After (4/4)**

```
interface TrafficLight {
  // ...
}
```

This refactoring pattern in itself does not add much value, but it enables fantastic improvements later. Having `is` methods for all the values is a smell, too, so we have replaced one smell with another. But we can handle these methods one by one, whereas the enum values were tightly connected. It is important to note that most of the `is` methods are temporary and do not exist for long — in this case, we get rid of some of them in this chapter and many more in chapter 5.

#### FURTHER READING

This refactoring pattern can also be found in Martin Fowler's book *Refactoring*.

### 4.1.4 Pushing code into classes

Now the magic is about to happen. All conditions in `handleInput` have to do with the `input` parameter, which means the code should be in that class. Luckily, there is a simple way to do this:

1. Make a new method in the `Input` interface, and give it a slightly different name than the source method `handleInput`. In this case, we are already in `Input`, so there is no point in writing it twice.

**Listing 4.29. New interface**

```
interface Input {
  // ...
  handle(): void;
}
```

2. Copy `handleInput`, and paste it into all the classes. Remove **`function`**, because it is now a method; and replace the `input` parameter with `this`. It still has the wrong name, so we still get errors.

**Listing 4.30. After**

```
class Right implements Input {
  // ...
  handleInput() { ❶
    if (this.isLeft()) ❷
      moveHorizontal(-1);
    else if (this.isRight()) ❷
      moveHorizontal(1);
    else if (this.isUp()) ❷
      moveVertical(-1);
    else if (this.isDown()) ❷
      moveVertical(1);
  }
}
```

❶ Remove **`function`** and the parameter.

❷ Change `input` to `this`.

3. Go through the `handleInput` methods in all four classes. The process is identical, so we show only one:
   a. Inline the return values of the methods `isLeft`, `isRight`, `isUp`, and `isDown`.

**Listing 4.31. Before**

```
class Right implements Input {
  // ...
  handleInput() {
    if (this.isLeft())
      moveHorizontal(-1);
    else if (this.isRight())
      moveHorizontal(1);
    else if (this.isUp())
      moveVertical(-1);
    else if (this.isDown())
      moveVertical(1);
  }
}
```

**Listing 4.32. After (1/4)**

```
class Right implements Input {
  // ...
  handleInput() {
    if (false) ❶
      moveHorizontal(-1);
    else if (true) ❶
      moveHorizontal(1);
    else if (false) ❶
      moveVertical(-1);
    else if (false) ❶
      moveVertical(1);
  }
}
```

❶ After inlining `is` methods

   b. Remove all the **`if (false) { … }`** and the **`if`** part of **`if (true)`**.

**Listing 4.33. Before**

```
class Right implements Input {
  // ...
  handleInput() {
    if (false)
      moveHorizontal(-1);
    else if (true)
      moveHorizontal(1);
    else if (false)
      moveVertical(-1);
    else if (false)
      moveVertical(1);
  }
}
```

**Listing 4.34. After (2/4)**

```
class Right implements Input {
  // ...
  handleInput() { moveHorizontal(1);
      }
}
```

c. Change the name to `handle`, to signal that we are finished with this method. The compiler should accept this method at this point.

**Listing 4.35. Before**

```
class Right implements Input {
  // ...
  handleInput() { moveHorizontal(1); }
}
```

**Listing 4.36. After (3/4)**

```
class Right implements Input {
  // ...
  handle() { moveHorizontal(1); }
}
```

4. Replace the body of `handleInput` with a call to our new method.

**Listing 4.37. Before**

```
function handleInput(input: Input) {
  if (input.isLeft())
    moveHorizontal(-1);
  else if (input.isRight())
    moveHorizontal(1);
  else if (input.isUp())
    moveVertical(-1);
  else if (input.isDown())
    moveVertical(1);
}
```

**Listing 4.38. After (4/4)**

```
function handleInput(input: Input) {
  input.handle();
}
```

After going through this process, we arrive at this nice improvement. All the **`if`**s are gone, and these methods easily fit in five lines.

**Listing 4.39. Before**

```
function handleInput(input: Input) {
  if (input.isLeft())
    moveHorizontal(-1);
  else if (input.isRight())
    moveHorizontal(1);
  else if (input.isUp())
    moveVertical(-1);
  else if (input.isDown())
    moveVertical(1);
}
```

**Listing 4.40. After**

```
function handleInput(input: Input) {
  input.handle();
}

interface Input {
  // ...
  handle(): void;
}
class Left implements Input {
  // ...
  handle() { moveHorizontal(-1); }
}
class Right implements Input {
  // ...
  handle() { moveHorizontal(1); }
}
class Up implements Input {
  // ...
  handle() { moveVertical(-1); }
}
class Down implements Input {
  // ...
  handle() { moveVertical(1); }
}
```

This is my favorite refactoring pattern: it is so structured that we can perform it with little cognitive load, but we end up with very nice code. I call it ***Push code into classes*** [P4.1.5].

### 4.1.5 *Refactoring:* Push code into classes

#### DESCRIPTION

This refactoring pattern is a natural continuation of ***Replace type code with classes*** [P4.1.3], as it moves functionality into classes. As a result, **`if`** statements are often eliminated, and functionality is moved closer to the data. As discussed earlier, this helps localize the invariants, because functionality connected with a specific value is moved into the class corresponding to that value.

In its simplest form, we always assume that we move an entire method into the classes. This is not a problem because, as we have seen, we usually start by extracting methods. It is possible to move code without extracting it first, but doing so requires more care to verify that we have not broken anything.

#### PROCESS

1. Make a new method in the target interface. Give it a slightly different name than the source method.
2. Copy the source function, and paste it into all the classes. Remove **`function`**, as it is now a method; replace the context with `this`; and remove the unused parameters. The method still has the wrong name, so we still get errors.
3. Go through the new method in all the classes:

a. Inline the methods that return a constant expression.
b. Perform all the computations we can up front, which usually amounts to removing `if (true)` and `if (false) { … }` but may also require simplifying the conditions first (for example, `false || true` becomes `true`).
c. Change the name to its proper name, to signal that we are finished with this method. The compiler should accept it.

4. Replace the body of the original function with a call to our new method.

### EXAMPLE

As this refactoring pattern is so closely related to ***Replace type code with classes*** [P4.1.3] we continue with the traffic light example.

**Listing 4.41. Initial**

```
interface TrafficLight {
  isRed(): boolean;
  isYellow(): boolean;
  isGreen(): boolean;
}
class Red implements TrafficLight {
  isRed() { return true; }
  isYellow() { return false; }
  isGreen() { return false; }
}
class Yellow implements TrafficLight {
  isRed() { return false; }
  isYellow() { return true; }
  isGreen() { return false; }
}
class Green implements TrafficLight {
  isRed() { return false; }
  isYellow() { return false; }
  isGreen() { return true; }
}
function updateCarForLight(current: TrafficLight) {
  if (current.isRed())
    car.stop();
  else
    car.drive();
}
```

We follow the process:

1. Make a new method in the target interface. Give it a slightly different name than the source method.

**Listing 4.42. New method**

```
interface TrafficLight {
  // ...
  updateCar(): boolean;
}
```

2. Copy the source function, and paste it into all the classes. Remove **`function`**, as it is now a method; replace the context with `this`; and remove the unused parameters. It still has the wrong name, so we still get errors.

**Listing 4.43. After (1/5)**

```
class Red implements TrafficLight {
  // ...
  updateCarForLight() {
    if (this.isRed())
      car.stop();
    else
      car.drive();
  }
}
class Yellow implements TrafficLight {
  // ...
  updateCarForLight() {
    if (this.isRed())
      car.stop();
    else
      car.drive();
  }
}
class Green implements TrafficLight {
  // ...
  updateCarForLight() {
    if (this.isRed())
      car.stop();
    else
      car.drive();
  }
}
```

3. Go through the new method in all the classes:
   a. Inline the methods that return a constant expression.

**Listing 4.44. Before**

```
class Red implements TrafficLight {
  // ...
  updateCarForLight() {
    if (this.isRed())
      car.stop();
    else
      car.drive();
  }
}
class Yellow implements TrafficLight
      {
  // ...
  updateCarForLight() {
    if (this.isRed())
      car.stop();
    else
      car.drive();
  }
}
class Green implements TrafficLight {
  // ...
  updateCarForLight() {
    if (this.isRed())
      car.stop();
    else
      car.drive();
  }
}
```

**Listing 4.45. After (2/5)**

```
class Red implements TrafficLight {
  // ...
  updateCarForLight() {
    if (true)
      car.stop();
    else
      car.drive();
  }
}
class Yellow implements TrafficLight
      {
  // ...
  updateCarForLight() {
    if (false)
      car.stop();
    else
      car.drive();
  }
}
class Green implements TrafficLight {
  // ...
  updateCarForLight() {
    if (false)
      car.stop();
    else
      car.drive();
  }
}
```

b. Perform all the computations we can up front.

**Listing 4.46. Before**

```
class Red implements TrafficLight {
  // ...
  updateCarForLight() {
    if (true)
      car.stop();
    else
      car.drive();
  }
}
class Yellow implements TrafficLight
      {
  // ...
  updateCarForLight() {
    if (false)
      car.stop();
    else
      car.drive();
  }
}
class Green implements TrafficLight {
  // ...
  updateCarForLight() {
    if (false)
      car.stop();
    else
      car.drive();
  }
}
```

**Listing 4.47. After (3/5)**

```
class Red implements TrafficLight {
  // ...
  updateCarForLight() { car.stop(); }
}
class Yellow implements TrafficLight
      {
  // ...
  updateCarForLight() { car.drive();
      }
}
class Green implements TrafficLight {
  // ...
  updateCarForLight() { car.drive();
      }
}
```

c. Change the name to its proper name, to signal that we are finished with this method.

**Listing 4.48. Before**

```
class Red implements TrafficLight {
  // ...
  updateCarForLight() { car.stop(); }
}
class Yellow implements TrafficLight
      {
  // ...
  updateCarForLight() { car.drive();
      }
}
class Green implements TrafficLight {
  // ...
  updateCarForLight() { car.drive();
      }
}
```

**Listing 4.49. After (4/5)**

```
class Red implements TrafficLight {
  // ...
  updateCar() { car.stop(); }
}
class Yellow implements TrafficLight
      {
  // ...
  updateCar() { car.drive(); }
}
class Green implements TrafficLight
      {
  // ...
  updateCar() { car.drive(); }
}
```

4. Replace the body of the original function with a call to our new method.

**Listing 4.50. Before**

```
function updateCarForLight(current:
      TrafficLight) {
  if (current.isRed())
    car.stop();
  else
    car.drive();
}
```

**Listing 4.51. After (5/5)**

```
function updateCarForLight(current:
      TrafficLight) {
  current.updateCar();
}
```

We mentioned earlier that the `is` methods become a smell if they remain, so it is worth noting that at this point we do not need any of them in this tiny example. This is an extension of the advantages of this refactoring pattern.

#### FURTHER READING

In this simple form, this refactoring is essentially the same as Martin Fowler's ***Move method***. However, I think this rebranding better conveys the intention and force behind it.

### 4.1.6 Inlining a superfluous method

At this point, we can see another amusing effect of refactoring. Even though we just introduced the `handleInput` function, that does not necessarily mean it should stay. Refactoring is often circular, adding things that enable further refactoring and then removing them again. So, never be afraid of adding code.

When we introduced `handleInput`, it had a clear purpose. Now, however, it does not add any readability to our program, and it takes up space, so we can remove it:

1. Change the method name to `handleInput2`. This makes the compiler error wherever we use the function.
2. Copy the body `input.handle();`, and note that `input` is the parameter.
3. We use this function in only one place, where we replace the call with the body.

**Listing 4.52. Before**

```
handleInput(current);
```

**Listing 4.53. After**

```
current.handle();
```

After this, and after a quick renaming of `current` to `input`, `handleInputs` looks like this.

**Listing 4.54. Before**

```
function handleInputs() {
  while (inputs.length > 0) {
    let current = inputs.pop();
    handleInput(current);
  }
}

function handleInput(input: Input) {
  input.handle();
}
```

**Listing 4.55. After**

```
function handleInputs() {
  while (inputs.length > 0) {
    let input = inputs.pop();
    input.handle(); (1)
  }
}

(2)
```

(1) Inlining method

(2) `handleInput` deleted

This refactoring pattern, ***Inline method*** [P4.1.7], is the exact inverse of ***Extract method*** [P3.2.1] from chapter 3.

### 4.1.7 *Refactoring:* Inline method

#### DESCRIPTION

Two great themes of this book are adding code (usually to support classes) and removing code. This refactoring pattern supports the latter: it removes methods that no longer add readability to our program. It does so by moving code from a method to all call sites. This makes the method unused, at which point we can safely delete it.

Notice that we differentiate between inlining methods and the refactoring pattern ***Inline method***. In the previous sections, we inlined the `is` methods while we were pushing code into classes, and then we used ***Inline method*** to eliminate the original function. When we inline methods (without the emphasis), we don't do it at every call site, so we preserve the original method. This is usually to simplify the call site. When we ***Inline method*** (emphasized), we do it at every call site and then delete the method.

In this book, we often do this when methods have only a single line. This is because of our strict five-line limit; inlining a method with a single line cannot break this rule. We can also apply this refactoring pattern to methods with more than one line.

Another consideration is whether the method is too complex to be inlined. The following method gives the absolute value of a number; we have optimized it for performance, so it is branch-free. It is a single line. It relies on low-level operations to achieve its purpose, so having the method adds readability, and we should not inline it. In this case, inlining it would also go against the smell "operations should be on the same level of abstraction," which motivated our ***Either call or pass*** [R3.3.1] rule.

**Listing 4.56. Method that should not be inlined**

```
const NUMBER_BITS = 32;
function absolute(x: number) {
  return (x ^ x >> NUMBER_BITS-1) - (x >> NUMBER_BITS-1);
}
```

### PROCESS

1. Change the method name to something temporary. This makes the compiler error wherever we use the function.
2. Copy the body of the method, and note its parameters.
3. Wherever the compiler gives errors, replace the call with the copied body, and map the arguments to the parameters.
4. Once we can compile without errors, we know the original method is unused. Delete the original method.

### EXAMPLE

In this example, we have discovered that we split the two parts of a bank transaction: withdrawing money from one account and depositing it into another. This means we can accidentally deposit money without withdrawing it, if we call the wrong method. To remedy the situation, we decide to join the two methods.

**Listing 4.57. Initial**

```
function deposit(to: string, amount: number) {
  let accountId = database.find(to);
  database.updateOne(accountId, { $inc: { balance: amount } });
}

function transfer(from: string, to: string, amount: number) {
  deposit(from, -amount);
  deposit(to, amount);
}
```

***In TypeScript . . .***

The symbol $ is treated like _. Thus it can be part of an identifier but has no special meaning.

We follow the process:

1. Change the method name to something temporary. This makes the compiler error wherever we use the function.

**Listing 4.58. Before**

```
function deposit(to: string,
    amount: number) {
  // ...
}
```

**Listing 4.59. After (1/2)**

```
function deposit2(to: string,
    amount: number) {
  // ...
}
```

2. Copy the body of the method, and note its parameters.
3. Wherever the compiler gives errors, replace the call with the copied body, and map the arguments to the parameters.

**Listing 4.60. Before**

```
function transfer(from: string, to:
      string,
    amount: number) {
  deposit(from, -amount);
  deposit(to, amount);
}
```

**Listing 4.61. After (2/2)**

```
function transfer(from: string, to:
      string,
    amount: number) {
  let fromAccountId =
      database.find(from);
  database.updateOne(fromAccountId,
    { $inc: { balance: -amount } });
  let toAccountId =
      database.find(to);
  database.updateOne(toAccountId,
    { $inc: { balance: amount } });
}
```

4. Once we can compile without errors, we know the original method is unused. Delete the original method.

At this point, money cannot be created from nothing in the code. It is debatable whether having this code duplication is a bad idea; in chapter 6, we see another solution that uses encapsulation.

#### FURTHER READING

This refactoring pattern can be found in Martin Fowler's book *Refactoring*.

## 4.2 A large if statement

Let's go through the same process, but with a bigger method: `drawMap`.

**Listing 4.62. Initial**

```
function drawMap(g: CanvasRenderingContext2D) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++) {
      if (map[y][x] === Tile.FLUX)
        g.fillStyle = "#ccffcc";
      else if (map[y][x] === Tile.UNBREAKABLE)
        g.fillStyle = "#999999";
      else if (map[y][x] === Tile.STONE || map[y][x] === Tile.FALLING_STONE)
        g.fillStyle = "#0000cc";
      else if (map[y][x] === Tile.BOX || map[y][x] === Tile.FALLING_BOX)
        g.fillStyle = "#8b4513";
      else if (map[y][x] === Tile.KEY1 || map[y][x] === Tile.LOCK1)
        g.fillStyle = "#ffcc00";
      else if (map[y][x] === Tile.KEY2 || map[y][x] === Tile.LOCK2)
        g.fillStyle = "#00ccff";

      if (map[y][x] !== Tile.AIR)
        g.fillRect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE, TILE_SIZE);
    }
  }
}
```

Immediately we notice a major violation of our ***`if` only at the start*** [R3.5.1] rule from the last chapter: there is a long **`else if`** chain right in the middle of the code. So, the

first thing we do is extract the **`else if`** chain to its own method.

**Listing 4.63. Before**

```
function drawMap(g:
      CanvasRenderingContext2D) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length;
      x++) {
      if (map[y][x] === Tile.FLUX)
        g.fillStyle = "#ccffcc";
      else if (map[y][x] ===
      Tile.UNBREAKABLE)
        g.fillStyle = "#999999";
      else if (map[y][x] === Tile.STONE
            || map[y][x] ===
      Tile.FALLING_STONE)
        g.fillStyle = "#0000cc";
      else if (map[y][x] === Tile.BOX
            || map[y][x] ===
      Tile.FALLING_BOX)
        g.fillStyle = "#8b4513";
      else if (map[y][x] === Tile.KEY1
            || map[y][x] === Tile.LOCK1)
        g.fillStyle = "#ffcc00";
      else if (map[y][x] === Tile.KEY2
            || map[y][x] === Tile.LOCK2)
        g.fillStyle = "#00ccff";

      if (map[y][x] !== Tile.AIR)
        g.fillRect(
          x * TILE_SIZE,
          y * TILE_SIZE,
          TILE_SIZE,
          TILE_SIZE);
    }
  }
}
```

**Listing 4.64. After *Extract method* [P3.2.1]**

```
function drawMap(g:
      CanvasRenderingContext2D) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length;
      x++) {
      colorOfTile(g, x, y); (1)
      if (map[y][x] !== Tile.AIR)
        g.fillRect(
          x * TILE_SIZE,
          y * TILE_SIZE,
          TILE_SIZE,
          TILE_SIZE);
    }
  }
}

function colorOfTile(g:
      CanvasRenderingContext2D,
    x: number, y: number) { (1)
  if (map[y][x] === Tile.FLUX)
    g.fillStyle = "#ccffcc";
  else if (map[y][x] ===
      Tile.UNBREAKABLE)
    g.fillStyle = "#999999";
  else if (map[y][x] === Tile.STONE
        || map[y][x] ===
      Tile.FALLING_STONE)
    g.fillStyle = "#0000cc";
  else if (map[y][x] === Tile.BOX
        || map[y][x] ===
      Tile.FALLING_BOX)
    g.fillStyle = "#8b4513";
  else if (map[y][x] === Tile.KEY1
        || map[y][x] === Tile.LOCK1)
    g.fillStyle = "#ffcc00";
  else if (map[y][x] === Tile.KEY2
        || map[y][x] === Tile.LOCK2)
    g.fillStyle = "#00ccff";
}
```

(1) Extracted method and call

For now, `drawMap` complies with our ***Five lines*** [R3.1.1] rule, so we continue with `colorOfTile`. `colorOfTile` violates ***Never use `if` with `else`*** [R4.1.1]. As we did earlier, to solve this issue, we replace the `Tile` enum with a `Tile` interface:

1. Introduce a new interface with the temporary name `Tile2`, with methods for all the values in our enum.

**Listing 4.65. New interface**

```
interface Tile2 {
  isFlux(): boolean;
```

```
  isUnbreakable(): boolean;
  isStone(): boolean;
  // ... (1)
}
```

(1) Methods for all the values of the enum

2. Create classes corresponding to each of the enum values.

**Listing 4.66. New classes**

```
class Flux implements Tile2 {
  isFlux() { return true; }
  isUnbreakable() { return false; }
  isStone() { return false; }
  // ...
}
class Unbreakable implements Tile2 { ... }
class Stone implements Tile2 { ... }
/// ... (1)
```

(1) Similar classes for the rest of the values of the enum

3. Rename the enum to `RawTile`, making the compiler show us wherever it is used.

**Listing 4.67. Before**

```
enum Tile {
  AIR,
  FLUX,
  UNBREAKABLE,
  PLAYER,
  STONE, FALLING_STONE,
  BOX, FALLING_BOX,
  KEY1, LOCK1,
  KEY2, LOCK2
}
```

**Listing 4.68. After (1/2)**

```
enum RawTile { (1)
  AIR,
  FLUX,
  UNBREAKABLE,
  PLAYER,
  STONE, FALLING_STONE,
  BOX, FALLING_BOX,
  KEY1, LOCK1,
  KEY2, LOCK2
}
```

(1) Changing the name to get compile errors

4. Replace equality checks with the new methods. We have to make this change in a lot of places throughout the application; here we show only `colorOfTile`.

**Listing 4.69. Before**

```
function colorOfTile(g:
      CanvasRenderingContext2D,
    x: number, y: number) {
  if (map[y][x] === Tile.FLUX)
    g.fillStyle = "#ccffcc";
  else if (map[y][x] === Tile.UNBREAKABLE)
    g.fillStyle = "#999999";
  else if (map[y][x] === Tile.STONE
        || map[y][x] === Tile.FALLING_STONE)
    g.fillStyle = "#0000cc";
  else if (map[y][x] === Tile.BOX
        || map[y][x] === Tile.FALLING_BOX)
    g.fillStyle = "#8b4513";
  else if (map[y][x] === Tile.KEY1
        || map[y][x] === Tile.LOCK1)
    g.fillStyle = "#ffcc00";
  else if (map[y][x] === Tile.KEY2
        || map[y][x] === Tile.LOCK2)
    g.fillStyle = "#00ccff";
}
```

**Listing 4.70. After (2/2)**

```
function colorOfTile(g:
      CanvasRenderingContext2D,
    x: number, y: number) {
  if (map[y][x].isFlux())          (1)
    g.fillStyle = "#ccffcc";
  else if (map[y][x].isUnbreakable()) (1)
    g.fillStyle = "#999999";
  else if (map[y][x].isStone()     (1)
        || map[y][x].isFallingStone()) (1)
    g.fillStyle = "#0000cc";
  else if (map[y][x].isBox()       (1)
        || map[y][x].isFallingBox()) (1)
    g.fillStyle = "#8b4513";
  else if (map[y][x].isKey1()      (1)
        || map[y][x].isLock1()) (1)
    g.fillStyle = "#ffcc00";
  else if (map[y][x].isKey2()      (1)
        || map[y][x].isLock2()) (1)
    g.fillStyle = "#00ccff";
}
```

(1) Use new methods instead of equality checks.

**WARNING** **Take care that `map[y][x] === Tile.FLUX` becomes `map[y][x].isFlux()`, and `map[y][x] !== Tile.AIR` becomes `!map[y][x].isAir()`. Pay attention to the `!`.**

5. Replace uses of `Tile.FLUX` with **`new`** `Flux()`, `Tile.AIR` with **`new`** `Air()`, and so forth.

At this point last time, we had no errors and could rename the temporary `Tile2` to the permanent `Tile`. This time the situation is different: we still have two places with errors showing that we are using `Tile`. This is why we use a temporary name; otherwise, we probably would not have spotted the issue in `remove` and would have assumed it was working — which it is not.

**Listing 4.71. Last two errors**

```
let map: Tile[][] = [ (1)
  [2, 2, 2, 2, 2, 2, 2, 2],
  [2, 3, 0, 1, 1, 2, 0, 2],
  [2, 4, 2, 6, 1, 2, 0, 2],
  [2, 8, 4, 1, 1, 2, 0, 2],
  [2, 4, 1, 1, 1, 9, 0, 2],
  [2, 2, 2, 2, 2, 2, 2, 2],
];

function remove(tile: Tile) { (1)
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++) {
      if (map[y][x] === tile) {
        map[y][x] = new Air();
      }
    }
```

```
  }
}
```

❶ Errors because we refer to `Tile`

Both of these errors require special treatment, so we go through them in turn.

### 4.2.1 Removing generality

The problem with `remove` is that it takes a tile type and removes it from everywhere on the map. That is, it does not check against a specific instance of `Tile`; instead, it checks that the instances are similar.

**Listing 4.72. Initial**

```
function remove(tile: Tile) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++) {
      if (map[y][x] === tile) {
        map[y][x] = new Air();
      }
    }
  }
}
```

In other words, the problem is that `remove` is too general. It can remove any type of tile. This generality makes it less flexible and more difficult to change. Therefore, we prefer specialization: we make a less general version and switch to using that, instead.

Before we can make a general version, we need to investigate how it is used. We want to make the parameter less general, so we look for what arguments are passed to it in practice. We use our familiar process and rename `remove` to a temporary name, `remove2`. We find that `remove` is used in four places.

**Listing 4.73. Before**

```
/// ...
remove(new Lock1());
/// ...
remove(new Lock2());
/// ...
remove(new Lock1());
/// ...
remove(new Lock2());
/// ...
```

We can see that even though `remove` supports removing any type of tile, in practice it is only removing `Lock1` or `Lock2`. We can take advantage of this:

1. Duplicate `remove2`.

**Listing 4.74. Before**

```
function remove2(tile: Tile) {
  // ...
}
```

**Listing 4.75. After (1/4)**

```
function remove2(tile: Tile) {
  // ... (1)
}
function remove2(tile: Tile) {
  // ... (1)
}
```

(1) They have the same body.

2. Rename one of them to `removeLock1`, remove its parameter, and replace `=== tile` with `=== Tile.LOCK1`.

**Listing 4.76. Before**

```
function remove2(tile: Tile) {
  for (let y = 0; y < map.length;
       y++) {
    for (let x = 0; x <
      map[y].length; x++) {
      if (map[y][x] === tile) {
        map[y][x] = new Air();
      }
    }
  }
}
```

**Listing 4.77. After (2/4)**

```
function removeLock1() { (1)
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length;
       x++) {
      if (map[y][x] === Tile.LOCK1) { (2)
        map[y][x] = new Air();
      }
    }
  }
}
```

(1) Rename and remove parameter

(2) Replace `tile` with `Tile.LOCK1`

3. This is exactly the type of equality we know how to eliminate, so as we did before, we replace it with the method call.

**Listing 4.78. Before**

```
function removeLock1() {
  for (let y = 0; y < map.length;
       y++) {
    for (let x = 0; x <
      map[y].length; x++) {
      if (map[y][x] === Tile.LOCK1)
      {
        map[y][x] = new Air();
      }
    }
  }
}
```

**Listing 4.79. After (3/4)**

```
function removeLock1() {
  for (let y = 0; y < map.length; y++)
      {
    for (let x = 0; x < map[y].length;
       x++) {
      if (map[y][x].isLock1()) { (1)
        map[y][x] = new Air();
      }
    }
  }
}
```

(1)

4. This function has no more errors, so we can switch the old calls over to use the new ones.

**Listing 4.80. Before**

```
remove(new Lock1());
```

**Listing 4.81. After (4/4)**

```
removeLock1();
```

We do the same thing for `removeLock2`. After that, we have `removeLock1` and `removeLock2` with no errors. `remove2` still has an error, but it is no longer called, so we simply delete it. In total, we performed this change.

**Listing 4.82. Before**

```
function remove(tile: Tile) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length;
      x++) {
      if (map[y][x] === tile) {
        map[y][x] = new Air();
      }
    }
  }
}
```

**Listing 4.83. After**

```
function removeLock1() {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length;
      x++) {
      if (map[y][x].isLock1()) {
        map[y][x] = new Air();
      }
    }
  }
}
function removeLock2() {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length;
      x++) {
      if (map[y][x].isLock2()) {
        map[y][x] = new Air();
      }
    }
  }
}
(1)
```

(1) Original `remove` is deleted

We call the process of introducing less-general versions of a function ***Specialize method*** [P4.2.2].

### 4.2.2 *Refactoring:* Specialize method

#### DESCRIPTION

This refactoring is more esoteric because it goes against the instincts of many programmers. We have a natural desire to generalize and reuse, but doing so can be problematic because it blurs responsibilities and means our code can be called from a variety of places. This refactoring pattern reverses these effects.

More specialized methods are called from fewer places, which means they become unused sooner, so we can remove them.

#### PROCESS

1. Duplicate the method we want to specialize.
2. Rename one of them to a new permanent name, and remove (or replace) the parameter we are using as the basis of our specialization.
3. Correct the method accordingly, so it has no errors.
4. Switch the old calls over to use the new ones.

#### EXAMPLE

Imagine that we are implementing a chess game. As part of our move-checker, we have come up with a brilliantly general expression to test whether a move fits a piece's pattern.

**Listing 4.84. Initial**

```
function canMove(start: Tile, end: Tile, dx: number, dy: number) {
  return dx * abs(start.x - end.x) === dy * abs(start.y - end.y)
      || dy * abs(start.x - end.x) === dx * abs(start.y - end.y);
}

/// ...
  if (canMove(start, end, 1, 0)) // Rook
/// ...
  if (canMove(start, end, 1, 1)) // Bishop
/// ...
  if (canMove(start, end, 1, 2)) // Knight
/// ...
```

We follow the process:

1. Duplicate the method we want to specialize.

**Listing 4.85. Before**

```
function canMove(start: Tile, end:
      Tile, dx: number, dy: number) {
  return dx * abs(start.x - end.x)
     === dy * abs(start.y - end.y)
      || dy * abs(start.x - end.x)
     === dx * abs(start.y - end.y);
}
```

**Listing 4.86. After (1/4)**

```
function canMove(start: Tile, end:
      Tile, dx: number, dy: number) {
  return dx * abs(start.x - end.x)
     === dy * abs(start.y - end.y)
      || dy * abs(start.x - end.x)
     === dx * abs(start.y - end.y);
}
function canMove(start: Tile, end:
      Tile, dx: number, dy: number) {
  return dx * abs(start.x - end.x)
     === dy * abs(start.y - end.y)
      || dy * abs(start.x - end.x)
     === dx * abs(start.y - end.y);
}
```

2. Rename one of them to a new permanent name, and remove (or replace) the parameter(s) we are specializing according to.

**Listing 4.87. Before**

```
function canMove(start: Tile, end: Tile,
      dx: number, dy: number) {
  return dx * abs(start.x - end.x)
     === dy * abs(start.y - end.y)
      || dy * abs(start.x - end.x)
     === dx * abs(start.y - end.y);
}
```

**Listing 4.88. After (2/4)**

```
function rookCanMove(start:
      Tile, end: Tile) {
  return 1 * abs(start.x -
      end.x)
     === 0 * abs(start.y -
      end.y)
      || 0 * abs(start.x -
      end.x)
     === 1 * abs(start.y -
      end.y);
}
```

3. Correct the method accordingly, so it has no errors.

**Listing 4.89. Before**

```
function rookCanMove(start: Tile,
      end: Tile) {
  return 1 * abs(start.x - end.x)
     === 0 * abs(start.y - end.y)
      || 0 * abs(start.x - end.x)
     === 1 * abs(start.y - end.y);
}
```

**Listing 4.90. After (3/4)**

```
function rookCanMove(start: Tile,
      end: Tile) {
  return abs(start.x - end.x) === 0
      || 0 === abs(start.y - end.y);
}
```

4. Switch the old calls over to use the new ones.

**Listing 4.91. Before**

```
  if (canMove(start, end, 1, 0)) //
      Rook
```

**Listing 4.92. After (4/4)**

```
  if (rookCanMove(start, end))
```

Notice that we no longer need the comment. `rookCanMove` is also much easier to understand: a rook can make a move if the change on either `x` or `y` is zero. We could even remove the `abs` part to simplify further.

I leave it to you to perform the same refactoring for the other pieces in the initial code. Are their methods as easy to understand?

#### FURTHER READING

To my knowledge, this is the first description of this as a refactoring pattern, although Jonathan Blow discussed the advantages of specialized methods versus general ones in his speech "How to program independent games" at UC Berkeley's Computer Science Undergraduate Association.

### 4.2.3 The only switch allowed

Only one error remains: we create our map using the enum indices, which no longer works. It is not difficult to imagine that we have already used these indices to store levels in files or to do something else. In practice, it is often not possible to change all the data to accommodate refactoring. So instead of changing the entire map, it is better to make a new function to take us from enum indices to the new classes. Luckily, this is straightforward to implement.

**Listing 4.93. Introducing `transformTile`**

```
let rawMap: RawTile[][] = [
  [2, 2, 2, 2, 2, 2, 2, 2],
  [2, 3, 0, 1, 1, 2, 0, 2],
  [2, 4, 2, 6, 1, 2, 0, 2],
  [2, 8, 4, 1, 1, 2, 0, 2],
  [2, 4, 1, 1, 1, 9, 0, 2],
  [2, 2, 2, 2, 2, 2, 2, 2],
];
let map: Tile2[][];
function assertExhausted(x: never): never { ❹
  throw new Error("Unexpected object: " + x);
}
```

```
function transformTile(tile: RawTile) { ❶
  switch (tile) {
    case RawTile.AIR: return new Air();
    case RawTile.PLAYER: return new Player();
    case RawTile.UNBREAKABLE: return new Unbreakable();
    case RawTile.STONE: return new Stone();
    case RawTile.FALLING_STONE: return new FallingStone();
    case RawTile.BOX: return new Box();
    case RawTile.FALLING_BOX: return new FallingBox();
    case RawTile.FLUX: return new Flux();
    case RawTile.KEY1: return new Key1();
    case RawTile.LOCK1: return new Lock1();
    case RawTile.KEY2: return new Key2();
    case RawTile.LOCK2: return new Lock2();
    default: assertExhausted(tile); ❹
  }
}
function transformMap() { ❷
  map = new Array(rawMap.length);
  for (let y = 0; y < rawMap.length; y++) {
    map[y] = new Array(rawMap[y].length);
    for (let x = 0; x < rawMap[y].length; x++) {
      map[y][x] = transformTile(rawMap[y][x]);
    }
  }
}
window.onload = () => {
  transformMap(); ❸
  gameLoop();
}
```

❶ New method for transforming a `RawTile` enum into a `Tile2` object

❷ New method for mapping the entire map

❸ Remember to call the new method.

❹ TypeScript trick explained shortly

***In TypeScript...***

An enum is a name for a number, as in C#, not classes, as in Java. So, we do not need any conversion between numbers and enums, and we can simply use the enum indices as in the previous code.

`TransformMap` exactly fits within our five-line limit.

With that, our application compiles without error. Now we can check that the game still works, rename `Tile2` to `Tile` everywhere, and commit our changes.

`transformTile` violates our five-line rule. It also almost violates another rule, ***Never use `switch`*** [R4.2.4], but we narrowly fall into the exception.

### 4.2.4 *Rule:* Never use `switch`

#### STATEMENT

Never use **`switch`** unless you have no **`default`** and return in *every* **`case`**.

### EXPLANATION

Switches are evil, as they allow for two "conveniences," each of which leads to bugs. First, when we do case analysis with **`switch`**, we do not always have to do something for every value; **`switch`** supports **`default`** for this purpose. With **`default`**, we can address many values without duplication. What we handle and what we don't is now invariant. However, like any default value, this stops the compiler from asking us to revalidate this invariant when we add a new value. To the compiler, there is no difference between us forgetting to handle a new value, and us wanting it to fall under **`default`**.

The other unfortunate convenience of **`switch`** is fall-through logic, where our program continues executing cases until it hits a **`break`**. It is easy to forget to include it and to not notice **`break`** is missing.

In general, I strongly recommend staying away from **`switch`**. But as specified in the detailed statement of the rule, we can remedy these maladies. The first way is easy: don't put functionality in **`default`**. In most languages, we should not have a **`default`**. Not all languages allow omitting the **`default`**, and if the language we are using doesn't, we should not use **`switch`** at all.

We address the fall-through concern by returning in every case: as a result, there is no fall-through, so there is no **`break`** to overlook.

***In TypeScript...***

Switches are particularly helpful, as we can make the compiler check that we have mapped all the enum values. We do need to introduce a "magic function" to make this work, but it is TypeScript-specific, so why it works is out of scope for this book. Luckily, the function never changes, and this pattern always works in TypeScript.

**Listing 4.94. Introducing `transformTile`**

```
function assertExhausted(x: never): never {
  throw new Error("Unexpected object: " + x);
}
/// ...
  switch (t) {
    case ...: return ...;
    // ...
    default: assertExhausted(t);
  }
```

This type of function is also one of the few that we cannot transform to fit in five lines if we want the compiler to check that we have mapped all the values.

### SMELL

In Martin Fowler's book *Refactoring*, **`switch`** is the name of a smell. `Switch` focuses on context: how to handle value X *here*. In contrast, pushing functionality into classes focuses on data: how this value (object) handles situation X. Focusing on context

means moving invariants further from their data, thereby globalizing the invariants.

#### INTENT

An elegant side-effect of this rule is that we transform **`switch`**es to **`else if`** chains, which we then make into classes, push code eliminating the **`if`**s, and in the end they disappear while preserving the functionality and making it easier and safer to add new values.

#### REFERENCES

As mentioned earlier, you can read more about the smell in Martin Fowler's book *Refactoring*.

### 4.2.5 Eliminating the if

Where were we? We are working on the `colorOfTile` function, and here is how it currently looks.

**Listing 4.95. Initial**

```
function colorOfTile(g: CanvasRenderingContext2D, x: number, y: number) {
  if (map[y][x].isFlux())
    g.fillStyle = "#ccffcc";
  else if (map[y][x].isUnbreakable())
    g.fillStyle = "#999999";
  else if (map[y][x].isStone()
        || map[y][x].isFallingStone())
    g.fillStyle = "#0000cc";
  else if (map[y][x].isBox()
        || map[y][x].isFallingBox())
    g.fillStyle = "#8b4513";
  else if (map[y][x].isKey1()
        || map[y][x].isLock1())
    g.fillStyle = "#ffcc00";
  else if (map[y][x].isKey2()
        || map[y][x].isLock2())
    g.fillStyle = "#00ccff";
}
```

`colorOfTile` violates the rule ***Never use `if` with `else`*** [R4.1.1]. We see that all the conditions in `colorOfTile` look at `map[y][x]`. This is the same condition we had earlier, so as before, we apply ***Push code into classes*** [P4.1.5]:

1. Make a `color` method in the `Tile` interface.
2. Copy `colorOfTile`, and paste it into all the classes. Remove **`function`**; in this case, remove the parameters `y` and `x`; and replace `map[y][x]` with `this`.
3. Go through the new method in all classes:
   a. Inline all the `is` methods.
   b. Remove **`if (true)`** and **`if (false) { … }`**. Most of them are left with a single line, and `Air` is empty.
   c. Change the name to `color`, to signal that we are finished with this method.
4. Replace the body of `colorOfTile` with a call to `map[y][x].color`.

At this point, the **`if`** is gone, and we are no longer violating any rules.

**Listing 4.96. Before**

```
function colorOfTile(g:
      CanvasRenderingContext2D,
    x: number, y: number) {
  if (map[y][x].isFlux())
    g.fillStyle = "#ccffcc";
  else if
      (map[y][x].isUnbreakable())
    g.fillStyle = "#999999";
  else if (map[y][x].isStone()
        ||
      map[y][x].isFallingStone())
    g.fillStyle = "#0000cc";
  else if (map[y][x].isBox()
        ||
      map[y][x].isFallingBox())
    g.fillStyle = "#8b4513";
  else if (map[y][x].isKey1()
        || map[y][x].isLock1())
    g.fillStyle = "#ffcc00";
  else if (map[y][x].isKey2()
        || map[y][x].isLock2())
    g.fillStyle = "#00ccff";
}
```

**Listing 4.97. After**

```
function colorOfTile(g:
      CanvasRenderingContext2D,
    x: number, y: number) {
  map[y][x].color(g);
}
interface Tile {
  // ...
  color(g: CanvasRenderingContext2D): void;
}
class Air implements Tile {
  // ...
  color(g: CanvasRenderingContext2D) {
    (1)
  }
}
class Flux implements Tile {
  // ...
  color(g: CanvasRenderingContext2D) {
    g.fillStyle = "#ccffcc"; (2)
  }
}
```

(1) `color` in `Air` is empty, as all the **`if`**s were **`false`**.

(2) All other classes have only their specific color.

`colorOfTile` has only a single line, so we decide to ***Inline method*** [P4.1.7]:

1. Change the method name to `colorOfTile2`.
2. Copy the body `map[y][x].color(g);`, and note that the parameters are `x`, `y`, and `g`.
3. We use this function in only one place, where we replace the call with the body.

**Listing 4.98. Before**

```
colorOfTile(g, x, y);
```

**Listing 4.99. After**

```
map[y][x].color(g);
```

In the end, we have the following.

**Listing 4.100. Before**

```
function drawMap(g:
      CanvasRenderingContext2D) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length;
      x++) {
      colorOfTile(g, x, y);
      if (map[y][x] !== Tile.AIR)
        g.fillRect(
          x * TILE_SIZE,
          y * TILE_SIZE,
          TILE_SIZE,
          TILE_SIZE);
    }
  }
}
function colorOfTile(g:
      CanvasRenderingContext2D,
    x: number, y: number) {
  map[y][x].color(g);
}
```

**Listing 4.101. After**

```
function drawMap(g:
      CanvasRenderingContext2D) {
  for (let y = 0; y < map.length; y++)
      {
    for (let x = 0; x < map[y].length;
      x++) {
      map[y][x].color(g); (2)
      if (!map[y][x].isAir())
        g.fillRect(
          x * TILE_SIZE,
          y * TILE_SIZE,
          TILE_SIZE,
          TILE_SIZE);
    }
  }
}
(1)
```

(1) `colorOfTile` is deleted.

(2) Inlined body

We have eliminated the large **`if`** from `draw`. But `draw` still does not comply with our rules, so we continue.

## 4.3 Code duplication

`drawMap` is in violation because it has an **`if`** in the middle. We can solve this by extracting the **`if`** as we have done many times. But this is the chapter of ***Push code into classes*** [P4.1.5], so we can also be adventurous and try that. This makes sense because both the **`if`** and the line before it concern `map[y][x]`.

**TIP** **If you want to be a bit daring, you can skip extracting the method and inlining it in the following process, and push it directly into the classes. Make sure you have committed first, so you can return to this point if something breaks.**

The procedure is the same as for `handleInput` and `colorOfTile`, except that we are not just extracting an **`if`**. We start with ***Extract method*** [P3.2.1] on the body of the **`for`**s.

**Listing 4.102. Before**

```
function drawMap(g:
      CanvasRenderingContext2D) {
  for (let y = 0; y < map.length;
      y++) {
    for (let x = 0; x <
      map[y].length; x++) {
      map[y][x].color(g);
      if (!map[y][x].isAir())
        g.fillRect(
          x * TILE_SIZE,
          y * TILE_SIZE,
          TILE_SIZE,
          TILE_SIZE);
    }
  }
}
```

**Listing 4.103. After**

```
function drawMap(g: CanvasRenderingContext2D)
      {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++) {
      drawTile(g, x, y);
    }
  }
}
function drawTile(g:
      CanvasRenderingContext2D, x: number, y:
      number) {
  map[y][x].color(g);
  if (!map[y][x].isAir())
    g.fillRect(
      x * TILE_SIZE,
      y * TILE_SIZE,
      TILE_SIZE,
      TILE_SIZE);
}
```

We can now use ***Push code into classes*** [P4.1.5] to move this method into the `Tile` classes.

**Listing 4.104. Before**

```
function drawTile(g:
      CanvasRenderingContext2D,
    x: number, y: number) {
  map[y][x].color(g);
  if (!map[y][x].isAir())
    g.fillRect(
      x * TILE_SIZE,
      y * TILE_SIZE,
      TILE_SIZE,
      TILE_SIZE);
}
```

**Listing 4.105. After**

```
function drawTile(g: CanvasRenderingContext2D,
    x: number, y: number) {
  map[y][x].draw(g, x, y);
}
interface Tile2 {
  // ...
  draw(g: CanvasRenderingContext2D,
    x: number, y: number): void;
}
class Air implements Tile {
  // ...
  draw(g: CanvasRenderingContext2D,
    x: number, y: number) {
    (1)
  }
}
class Flux implements Tile {
  // ...
  draw(g: CanvasRenderingContext2D,
    x: number, y: number) {
    g.fillStyle = "#ccffcc"; (2)
    g.fillRect( (2)
      x * TILE_SIZE, (2)
      y * TILE_SIZE, (2)
      TILE_SIZE, (2)
      TILE_SIZE); (2)
  }
}
```

(1) `draw` in `Air` ends up being empty.

(2) All other classes end up with two lines after inlining `color` and `isAir` and deleting the `if (true)`.

As usual, after we ***Push code into classes*** [P4.1.5], we have a function with only one line: `drawTile`. So, we use ***Inline method*** [P4.1.7].

**Listing 4.106. Before**

```
function drawMap(g:
      CanvasRenderingContext2D) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length;
      x++) {
      drawTile(g, x, y);
    }
  }
}
function drawTile(g:
      CanvasRenderingContext2D,
    x: number, y: number) {
  map[y][x].draw(g);
}
```

**Listing 4.107. After**

```
function drawMap(g:
      CanvasRenderingContext2D) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length;
      x++) {
      map[y][x].draw(g, x, y); (1)
    }
  }
}
(2)
```

(1) Inlined body

(2) `drawTile` is deleted.

At this point, you may be asking: what is up with all this code duplication in the classes? Couldn't we use an abstract class instead of the interface, and put all the

common code there? Let's answer each question in turn.

### 4.3.1 Couldn't we use an abstract class instead of the interface?

First of all, yes. Yes, we could do that, and it would avoid the code duplication. However, that approach also has some significant drawbacks. First, using an interface forces us to actively do something for each new class we introduce. Therefore, we cannot accidentally forget a property or override something we shouldn't. This is especially true six months from now when we have forgotten how this works and we return to add a new tile type.

This concept is so strong that it is also formalized in a rule, which prevents us from using abstract classes: ***Only inherit from interfaces*** [R4.3.2].

### 4.3.2 *Rule:* Only inherit from interfaces

#### STATEMENT

Only inherit from interfaces.

#### EXPLANATION

This rule simply states that we can only inherit from interfaces, as opposed to classes or abstract classes. The most common reason people use abstract classes is to provide a default implementation for some methods while having others be abstract. This reduces duplication and is convenient if we are lazy.

Unfortunately, the disadvantages are much more significant. Shared code causes coupling. In this case, the coupling is the code in the abstract class. Imagine that two methods are implemented in the abstract class: `methodA` and `methodB`. We find out that one subclass needs only `methodA` another needs only `methodB`. Our only option, in this case, is to override one of the methods with an empty version.

When we have a method with a default implementation, there are two scenarios: either every possible subclass needs the method, in which case we can easily move it out of the class; or *some* subclasses need to override it, but because it has an implementation, the compiler does not remind us of the method when we add a new subclass.

This is another instance of the issues with defaults, discussed earlier. In this case, it is better to leave methods entirely abstract, because then we need to explicitly handle these cases.

When multiple classes need to share code, we can put that code in another shared class. We return to this in chapter 5, when we discuss ***Introduce strategy-pattern*** [P5.4.2].

#### SMELL

I derived this rule from the principle "Favor object composition over inheritance" from the book *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [1]. This book also introduced the concept of *design patterns* to object-oriented programming.

#### INTENT

The smell states plainly that we should share code by referring other objects in favor of inheriting from them. This rule takes it to the extreme, as it is extremely rare for a problem to require inheritance; and when it doesn't, composition gives us a more flexible and stable solution.

#### REFERENCES

As mentioned, the rule comes from the book *Design Patterns*. We explore a better solution to get the desired code sharing in chapter 5, when we discuss the ***Introduce strategy-pattern*** [P5.4.2] refactoring.

### 4.3.3 What is up with all this code duplication?

In many cases, code duplication is bad. Everybody knows this, but let's think about why it is. Code duplication is bad when we need to maintain the code; we have to change something in a way that propagates throughout the program.

If we have duplicated code, and we change it in one place, we now have two different functions. Another way to say this is that code duplication is bad because it encourages divergence.

In most cases, that is not what you want; but in this particular case, it would be better. We expect that the graphics for different tiles should change over time and should be different. To make a point of this, consider how easy it would be to make the keys round.

If the code should have converged, how should we have dealt with it, when we cannot use inheritance? We return to this exact situation in the next chapter.

## 4.4 A pair of complex if statements

The next two functions that remain in violation of our rules are `moveHorizontal` and `moveVertical`. They are almost identical, so I present only the more complicated of the two, leaving the other as an exercise for you. `moveHorizontal` currently looks complicated; luckily, we can ignore most of it for now.

**Listing 4.108. Initial**

```
function moveHorizontal(dx: number) {
  if (map[playery][playerx + dx].isFlux()
          || map[playery][playerx + dx].isAir()) { ❶
    moveToTile(playerx + dx, playery);
  } else if ((map[playery][playerx + dx].isStone()
          || map[playery][playerx + dx].isBox()) ❶
          && map[playery][playerx + dx + dx].isAir()
          && !map[playery + 1][playerx + dx].isAir()) {
    map[playery][playerx + dx + dx] = map[playery][playerx + dx];
    moveToTile(playerx + dx, playery);
  } else if (map[playery][playerx + dx].isKey1()) {
```

```
    removeLock1();
    moveToTile(playerx + dx, playery);
  } else if (map[playery][playerx + dx].isKey2()) {
    removeLock2();
    moveToTile(playerx + dx, playery);
  }
}
```

❶ ||s that we want to preserve

First, notice that we have two ||s. These express something about the underlying domain. So, we would like to not only preserve this structure but emphasize it. We do so by pushing only that part into the classes.

This is a little different from what we have done before, as we are not pushing an entire method; however, the process stays the same. The most difficult part is coming up with a good name. This is the time to look at what the code is doing and be careful. We want to state that there is a relation between flux and air; it relates to the game and not something general, so we will not dwell on it but will simply say that they are *edible*:

1. Introduce an `isEdible` method in the interface.
2. In each class, add a method with a slightly wrong name: `isEdible2`.
3. As the body, put `return this.isFlux() || this.isAir();`.
4. Inline the values of `isFlux` and `isAir`.
5. Remove the temporary `2` in the name.
6. Replace `map[playery][playerx + dx].isFlux() || map[playery][playerx + dx].isAir()` *only here*. We cannot replace it everywhere because we do not know if other ||s refer to the same property.

The same situation is true for the other ||s. Here, boxes and stones share the property of being pushable in this context. Following the same pattern, we end up with the following code.

**Listing 4.109. Before**

```
function moveHorizontal(dx: number) {
  if (map[playery][playerx +
      dx].isFlux()
          || map[playery][playerx +
      dx].isAir()) {
    moveToTile(playerx + dx, playery);
  } else if ((map[playery][playerx +
      dx].isStone()
          || map[playery][playerx +
      dx].isBox())
          && map[playery][playerx + dx +
      dx].isAir()
          && !map[playery + 1][playerx +
      dx].isAir()) {
    map[playery][playerx + dx + dx] =
      map[playery][playerx + dx];
    moveToTile(playerx + dx, playery);
  } else if (map[playery][playerx +
      dx].isKey1()) {
    removeLock1();
    moveToTile(playerx + dx, playery);
  } else if (map[playery][playerx +
      dx].isKey2()) {
    removeLock2();
    moveToTile(playerx + dx, playery);
  }
}
```

**Listing 4.110. After**

```
function moveHorizontal(dx: number) {
  if (map[playery][playerx +
      dx].isEdible()) { (1)
    moveToTile(playerx + dx, playery);
  } else if (map[playery][playerx +
      dx].isPushable() (1)
          && map[playery][playerx + dx +
      dx].isAir()
          && !map[playery + 1][playerx +
      dx].isAir()) {
    map[playery][playerx + dx + dx] =
      map[playery][playerx + dx];
    moveToTile(playerx + dx, playery);
  } else if (map[playery][playerx +
      dx].isKey1()) {
    removeLock1();
    moveToTile(playerx + dx, playery);
  } else if (map[playery][playerx +
      dx].isKey2()) {
    removeLock2();
    moveToTile(playerx + dx, playery);
  }
}
interface Tile {
  // ...
  isEdible(): boolean;
  isPushable(): boolean;
}
class Box implements Tile { (2)
  // ...
  isEdible() { return false; }
  isPushable() { return true; }
}
class Air implements Tile { (3)
  // ...
  isEdible() { return true; }
  isPushable() { return false; }
}
```

(1) New helper methods

(2) `Box` and `Stone` are similar.

(3) `Air` and `Flux` are similar.

Having preserved the behavior of the `||`s, we move on as normal and look at the context. The context of this code is `map[playery][playerx + dx]`, as it is used in every **`if`**. Here we see that ***Push code into classes*** [P4.1.5] applies not only when we start with a series of equality checks but also to anything with a clear context — that is, a lot of `.`s with the same thing on the left. So, we push the code into `map[playery][playerx + dx]`; `Tile` again. After ***Push code into classes*** [P4.1.5], the code looks like this.

**Listing 4.111. Initial**

```
function moveHorizontal(dx: number) {
  if (map[playery][playerx +
      dx].isEdible()) {
    moveToTile(playerx + dx, playery);
  } else if (map[playery][playerx +
      dx].isPushable()
      && map[playery][playerx + dx +
      dx].isAir()
      && !map[playery + 1][playerx +
      dx].isAir()) {
    map[playery][playerx + dx + dx] =
      map[playery][playerx + dx];
    moveToTile(playerx + dx, playery);
  } else if (map[playery][playerx +
      dx].isKey1()) {
    removeLock1();
    moveToTile(playerx + dx, playery);
  } else if (map[playery][playerx +
      dx].isKey2()) {
    removeLock2();
    moveToTile(playerx + dx, playery);
  }
}
```

**Listing 4.112. After**

```
function moveHorizontal(dx: number) {
  map[playery][playerx +
      dx].moveHorizontal(dx);
}
interface Tile {
  // ...
  moveHorizontal(dx: number): void;
}
class Box implements Tile { (1)
  // ...
  moveHorizontal(dx: number) {
    if (map[playery][playerx + dx +
      dx].isAir()
        && !map[playery + 1][playerx +
      dx].isAir()) {
      map[playery][playerx + dx + dx]
      = this;
      moveToTile(playerx + dx,
      playery);
    }
  }
}
class Key1 implements Tile { (2)
  // ...
  moveHorizontal(dx: number) {
    removeLock1();
    moveToTile(playerx + dx, playery);
  }
}
class Lock1 implements Tile { (3)
  // ...
  moveHorizontal(dx: number) { }
}
class Air implements Tile { (4)
  // ...
  moveHorizontal(dx: number) {
    moveToTile(playerx + dx, playery);
  }
}
```

(1) `Box` and `Stone` are similar.

(2) `Key1` and `Key2` are similar.

(3) `Lock1` and `Lock2` are empty.

(4) `Air` and `Flux` are similar.

As usual, the original method is only a single line, so we inline it. Notice that because this **`if`** was more complex, there are artifacts from it in `Box` and `Stone`. Luckily, they still comply with our rules.

The only method that remains in conflict with our new rule ***Never use `if` with `else`*** [R4.1.1] is `updateTile`. But that method has a hidden structure, which we explore further in the next chapter.

## 4.5 Cleaning up

We end this chapter with some cleanup. We introduced a lot of new methods, and we deleted some after inlining them, but we can go further.

Many IDEs — including Visual Studio Code — indicate if a function is unused. Whenever we see this, and we are not in the middle of something, we should delete the function immediately. Deleting code saves us time because we don't have to deal with it in the future.

Unfortunately, because `interface`s are public, no IDE can tell you whether the methods in an interface are unused. We may intend to use them in the future, or they may be used by something outside of our scope. In general, we cannot easily delete methods from interfaces.

But the interfaces we have considered in this chapter were all introduced by us; therefore, we know the entire scope. We are free to do with them as we please: in particular, we can delete unused methods from them. Here is a technique for discovering whether methods are unused:

1. Compile. There should be no errors.
2. Delete a method from the interface.
3. Compile.
   a. If the compiler errors, undo, and move on.
   b. Otherwise, go through each class and check if you can delete the same method from it without getting errors.

This is a simple technique, but it is useful. After cleaning our interfaces, they have only 1 and 10 methods, respectively. I am such a big fan of deleting code that I have made a refactoring pattern out of this process: ***Try delete then compile*** [P4.5.1].

### 4.5.1 *Refactoring:* Try delete then compile

#### DESCRIPTION

This refactoring pattern's primary use is to remove unused methods from interfaces when we know their entire scope. We can also use this pattern to find and remove any unused methods. Performing ***Try delete then compile*** is as simple as the name describes: try deleting a method and see if the compiler allows it. This refactoring pattern is interesting not for its sophistication, but for its purpose. Note that we should not perform this refactoring while we are implementing new features, as we might delete methods that are not used *yet*.

Having expired code in a codebase drags it down. It takes time to read or ignore, and it makes compilation and analyses slower and testing more difficult. The quicker we can remove irrelevant code, the cheaper the process in terms of cost and effort.

To help identify unused methods, lots of editors highlight them in some way. But the analyses in these editors can be cheated. One of the things that can cheat the analyses is an interface. If a method is in an interface, it may be because the method needs to be

available for code outside of our scope or because we need the method for code inside our scope. Editors cannot tell the difference. The only safe option to them is to assume that all interface methods are meant to be used outside our scope.

When we know an interface is used only in our scope, we need to clean it up manually. This is the purpose of this refactoring pattern.

#### PROCESS

1. Compile. There should be no errors.
2. Delete a method from the interface.
3. Compile.
   a. If the compiler errors, undo, and move on.
   b. Otherwise, go through each class and check if you can delete the same method from it without getting errors.

#### EXAMPLE

In this artificial piece of code, there are three unused methods, but they are not all highlighted by the editor. In some editors, none are highlighted.

**Listing 4.113. Initial**

```
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();
```

Following the process, can you discover and eliminate the three unused methods?

## 4.6 Summary

- The rules ***Never use `if` with `else`*** [R4.1.1] and ***Never use `switch`*** [R4.2.4] state that we should have **`else`**s or **`switch`**es only at the edges of our program. Both **`else`**s and **`switch`**es are low-level control-flow operators. In the core of our applications, we should use the refactoring patterns ***Replace type code with classes*** [P4.1.3] and ***Push code into classes*** [P4.1.5] to replace **`switch`**es and **`else if`** chains with high-level classes and methods.
- Overly general methods can prevent us from refactoring. In these cases, we can use the refactoring pattern ***Specialize method*** [P4.2.2] to remove unnecessary generality.
- The rule ***Only inherit from interfaces*** [R4.3.2] prevents us from reusing code by using abstract classes and class inheritance because these types of inheritance impose unnecessarily tight coupling.

- We added two refactoring patterns for cleaning up after refactoring. ***Inline method*** [P4.1.7] and ***Try delete then compile*** [P4.5.1] can both remove methods that no longer add readability.

1. Often referred to as "The Gang of Four."

# 5 Unifying code to simplify and enable reuse

***This chapter covers:***

- Unifying similar classes with ***Unify similar classes*** [P5.1.1]
- Exposing structure with conditional arithmetic
- Understanding simple UML class diagrams
- Unifying similar code with ***Introduce strategy-pattern*** [P5.4.2]
- Removing clutter with ***No interface with only one implementation*** [R5.4.3]

In the last chapter, I mentioned that we are not finished with `updateTile`. It violates several rules, most notably ***Never use `if` with `else`*** [R4.1.1]. We also worked to preserve the `||`s in the code because they expressed structure. In this chapter, we explore how to expose more such structures in the code.

This is `updateTile` at the moment.

**Listing 5.1. Initial**

```
function updateTile(x: number, y: number) {
  if ((map[y][x].isStone() || map[y][x].isFallingStone())
      && map[y + 1][x].isAir()) {
    map[y + 1][x] = new FallingStone();
    map[y][x] = new Air();
  } else if ((map[y][x].isBox() || map[y][x].isFallingBox())
      && map[y + 1][x].isAir()) {
    map[y + 1][x] = new FallingBox();
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone()) {
    map[y][x] = new Stone();
  } else if (map[y][x].isFallingBox()) {
```

```
    map[y][x] = new Box();
  }
}
```

## 5.1 Unifying similar classes

The first thing we spot is that, as was the case earlier, we have parenthesized expressions (that is, `(map[y][x].isStone() || map[y][x].isFallingStone())`) that express a relation we want to not only preserve, we also emphasize. Therefore, our first step is to introduce one function for each of the two parenthesized `||`s. We say that `stony` and `boxy` should be understood as "behaves like a stone" and "behaves like a box," respectively.

**Listing 5.2. Before**

```
function updateTile(x: number, y:
      number) {
  if ((map[y][x].isStone()
         ||
      map[y][x].isFallingStone())
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new
      FallingStone();
    map[y][x] = new Air();
  } else if ((map[y][x].isBox()
         ||
      map[y][x].isFallingBox())
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new FallingBox();
    map[y][x] = new Air();
  } else if
      (map[y][x].isFallingStone()) {
    map[y][x] = new Stone();
  } else if
      (map[y][x].isFallingBox()) {
    map[y][x] = new Box();
  }
}
```

**Listing 5.3. After**

```
function updateTile(x: number, y: number) {
  if (map[y][x].isStony() (1)
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new FallingStone();
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy() (1)
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new FallingBox();
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone()) {
    map[y][x] = new Stone();
  } else if (map[y][x].isFallingBox()) {
    map[y][x] = new Box();
  }
}

interface Tile {
  // ...
  isStony(): boolean; (1)
  isBoxy(): boolean; (1)
}
/// In Stone and FallingStone, isStony
      returns true
/// In Box and FallingBox, isBoxy returns
      true
class Air implements Tile {
  // ...
  isStony() { return false; } (1)
  isBoxy() { return false; } (1)
}
```

(1) New helper methods

Having dealt with the `||`s, we can push the code into classes, but we can also wait and first take a look at the classes and the many methods we introduced in the last chapter. At this point, ***Try delete then compile*** [P4.5.1] let's us delete `isStone` and `isBox`.

We notice that the only difference between `Stone` and `FallingStone` is the result of the `isFallingStone` and the `moveHorizontal` methods.

**Listing 5.4. `Stone`**

```
class Stone implements Tile {
  isAir() { return false; }
  isFallingStone() { return false; } (1)
  isFallingBox() { return false; }
  isLock1() { return false; }
  isLock2() { return false; }
  draw(g: CanvasRenderingContext2D, x:
      number, y: number) {
    // ...
  }
  moveVertical(dy: number) { }
  isStony() { return true; }
  isBoxy() { return false; }
  moveHorizontal(dx: number) {
    // ... (1)
  }
}
```

(1) Only differences

**Listing 5.5. `FallingStone`**

```
class FallingStone implements Tile {
  isAir() { return false; }
  isFallingStone() { return true; } (1)
  isFallingBox() { return false; }
  isLock1() { return false; }
  isLock2() { return false; }
  draw(g: CanvasRenderingContext2D, x:
      number, y: number) {
    // ...
  }
  moveVertical(dy: number) { }
  isStony() { return true; }
  isBoxy() { return false; }
  moveHorizontal(dx: number) {
    (1)
  }
}
```

(1) Only differences

When a method returns a constant, we call it a *constant method*. We can join these two classes because they share a constant method that returns a different value in each case. Joining two classes like this happens in two phases, and the process is reminiscent of the algorithm for adding fractions. The first step in adding fraction is making the denominators equal, in the same way the first phase in joining classes is to make the classes equal in all but the constant methods. The second phase for fractions is the actual addition; for classes, it's the actual joining. Let's see how it looks in practice.

1. The first phase makes the two `moveHorizontal`s equal:
   a. In the body of each `moveHorizontal`, add an enclosing **`if (true) { }`** around the existing code.

**Listing 5.6. Before**

```
class Stone implements Tile {
  // ...
  moveHorizontal(dx: number) {
    if (map[playery][playerx + dx + dx].isAir()
      && !map[playery + 1][playerx + dx].isAir()) {
      map[playery][playerx + dx + dx] = this;
      moveToTile(playerx + dx, playery);
    }
  }
}
class FallingStone implements Tile {
  // ...
  moveHorizontal(dx: number) { }
}
```

**Listing 5.7. After (1/8)**

```
class Stone implements Tile {
  // ...
  moveHorizontal(dx: number) {
    if (true) { (1)
      if (map[playery][playerx + dx + dx].isAir()
        && !map[playery + 1][playerx + dx].isAir())
      {
        map[playery][playerx + dx + dx] = this;
        moveToTile(playerx + dx, playery);
      }
    }
  }
}
class FallingStone implements Tile {
  // ...
  moveHorizontal(dx: number) {
    if (true) { } (1)
  }
}
```

(1) New `if (true)`s

b. Replace **`true`** with `isFallingStone() === true` and `isFallingStone() === false`, respectively.

**Listing 5.8. Before**

```
class Stone implements Tile {
  // ...
  moveHorizontal(dx: number) {
    if (true) {
      if (map[playery][playerx + dx + dx].isAir()
        && !map[playery + 1][playerx + dx].isAir()) {
        map[playery][playerx + dx + dx] = this;
        moveToTile(playerx + dx, playery);
      }
    }
  }
}
class FallingStone implements Tile {
  // ...
  moveHorizontal(dx: number) {
    if (true) { }
  }
}
```

**Listing 5.9. After (2/8)**

```
class Stone implements Tile {
  // ...
  moveHorizontal(dx: number) {
    if (this.isFallingStone() === false) { (1)
      if (map[playery][playerx + dx + dx].isAir()
        && !map[playery + 1][playerx + dx].isAir())
      {
        map[playery][playerx + dx + dx] = this;
        moveToTile(playerx + dx, playery);
      }
    }
  }
}
class FallingStone implements Tile {
  // ...
  moveHorizontal(dx: number) {
    if (this.isFallingStone() === true) { } (1)
  }
}
```

(1) Specialized conditions

c. Copy the body of each `moveHorizontal` and paste it with an **`else`** into the other `moveHorizontal`.

**Listing 5.10. Before**

```
class Stone implements Tile {
  // ...
  moveHorizontal(dx: number) {
    if (this.isFallingStone() === false) {
      if (map[playery][playerx + dx + dx].isAir()
        && !map[playery + 1][playerx + dx].isAir()) {
        map[playery][playerx + dx + dx] = this;
        moveToTile(playerx + dx, playery);
      }
    }
  }
}
class FallingStone implements Tile {
  // ...
  moveHorizontal(dx: number) {
    if (this.isFallingStone() === true) { }
  }
}
```

**Listing 5.11. After (3/8)**

```
class Stone implements Tile {
  // ...
  moveHorizontal(dx: number) {
    if (this.isFallingStone() === false) {
      if (map[playery][playerx + dx + dx].isAir()
        && !map[playery + 1][playerx + dx].isAir())
      {
        map[playery][playerx + dx + dx] = this;
        moveToTile(playerx + dx, playery);
      }
    } else if (this.isFallingStone() === true) { (1)
    }
  }
}
class FallingStone implements Tile {
  // ...
  moveHorizontal(dx: number) {
    if (this.isFallingStone() === false) { (1)
      if (map[playery][playerx + dx + dx].isAir()
        && !map[playery + 1][playerx + dx].isAir())
      {
        map[playery][playerx + dx + dx] = this;
        moveToTile(playerx + dx, playery);
      }
    } else if (this.isFallingStone() === true) {
    }
  }
}
```

(1) Body from the other method

2. Now that only the `isFallingStone` constant methods are different, the second phase begins by introducing a `falling` field and assigning its value in the constructor.

**Listing 5.12. Before**

```
class Stone implements Tile {
  // ...
  isFallingStone() { return false; }
}
class FallingStone implements Tile {
  // ...
  isFallingStone() { return true; }
}
```

**Listing 5.13. After (4/8)**

```
class Stone implements Tile {
  private falling: boolean; (1)
  constructor() {
    this.falling = false; (2)
  }
  // ...
  isFallingStone() { return false; }
}
class FallingStone implements Tile {
  private falling: boolean; (1)
  constructor() {
    this.falling = true; (2)
  }
  // ...
  isFallingStone() { return true; }
}
```

(1) New field

(2) Assigns a default value to the new field

3. Change `isFallingStone` to return the new `falling` field.

**Listing 5.14. Before**

```
class Stone implements Tile {
  // ...
  isFallingStone() { return
      false; }
}
class FallingStone implements
      Tile {
  // ...
  isFallingStone() { return true;
      }
}
```

**Listing 5.15. After (5/8)**

```
class Stone implements Tile {
  // ...
  isFallingStone() { return this.falling;
      } (1)
}
class FallingStone implements Tile {
  // ...
  isFallingStone() { return this.falling;
      } (1)
}
```

(1) Returns a field instead of a constant

4. Compile to ensure that we have not broken anything yet.
5. For each of the classes:
   a. Copy the default value of `falling` and then make the default value a parameter.

**Listing 5.16. Before**

```
class Stone implements Tile {
  private falling: boolean;
  constructor() {
    this.falling = false;
  }
  // ...
}
```

**Listing 5.17. After (6/8)**

```
class Stone implements Tile {
  private falling: boolean;
  constructor(falling: boolean) { (1)
    this.falling = falling; (1)
  }
  // ...
}
```

(1) Makes `falling` a parameter

   b. Go through the compiler errors, and insert the default value as an argument.

**Listing 5.18. Before**

```
/// ...
  new Stone();
/// ...
```

**Listing 5.19. After (7/8)**

```
/// ...
  new Stone(false); (1)
/// ...
```

(1) Calls with the default value

6. Delete all but one of the classes we are unifying, and fix all of the compile errors by switching to the class that is still there.

**Listing 5.20. Before**

```
/// ...
  new FallingStone(true);
/// ...
```

**Listing 5.21. After (8/8)**

```
/// ...
  new Stone(true); (1)
/// ...
```

(1) Replaces the deleted class with the unified one

This unification amounts to the following transformation.

**Listing 5.22. Before**

```
function updateTile(x: number, y: number) {
  if (map[y][x].isStony()
          && map[y + 1][x].isAir()) {
    map[y + 1][x] = new FallingStone();
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
          && map[y + 1][x].isAir()) {
    map[y + 1][x] = new FallingBox();
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone()) {
    map[y][x] = new Stone();
  } else if (map[y][x].isFallingBox()) {
    map[y][x] = new Box();
  }
}
class Stone implements Tile {
  // ...
  isFallingStone() { return false; }
  moveHorizontal(dx: number) {
    if (map[playery][playerx + dx + dx].isAir()
      && !map[playery + 1][playerx + dx].isAir()) {
      map[playery][playerx + dx + dx] = this;
      moveToTile(playerx + dx, playery);
    }
  }
}
class FallingStone implements Tile {
  // ...
  isFallingStone() { return true; }
  moveHorizontal(dx: number) { }
}
```

**Listing 5.23. After**

```
function updateTile(x: number, y: number) {
  if (map[y][x].isStony()                          (1)
          && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Stone(true);               (1)
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()                    (1)
          && map[y + 1][x].isAir()) {
    map[y + 1][x] = new FallingBox();
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone()) {
    map[y][x] = new Stone(false);                  (1)
  } else if (map[y][x].isFallingBox()) {
    map[y][x] = new Box();
  }
}
class Stone implements Tile {
  constructor(private falling: boolean) {}         (1)
  // ...
  isFallingStone() { return this.falling; }        (2)
  moveHorizontal(dx: number) {                     (4)
    if (this.isFallingStone() === false) {
      if (map[playery][playerx + dx + dx].isAir()
        && !map[playery + 1][playerx + dx].isAir()) {
        map[playery][playerx + dx + dx] = this;
        moveToTile(playerx + dx, playery);
      }
    } else if (this.isFallingStone() === true) {
    }
  }
}
(3)
```

(1) Private field, set in the constructor

(2) `isFallingStone` returns this field.

(3) `FallingStone` is removed.

(4) `moveHorizontal` has the combined bodies.

***In TypeScript...***

Constructors behave a little differently than in most languages. First, we can have only one constructor, and it is always called **`constructor`**.

Second, putting **`public`** or **`private`** in front of a parameter to the constructor automatically makes an instance variable and assign it with the value of the argument. So these are equivalent:

**Listing 5.24. Before**

```
class Stone implements Tile {
  private falling: boolean;
  constructor(falling: boolean) {
    this.falling = falling;
  }
}
```

**Listing 5.25. After**

```
class Stone implements Tile {
  constructor(private falling: boolean) {}
}
```

We generally prefer the more compact one in this book.

Looking at the resulting `moveHorizontal`, we spot multiple interesting points. Most obvious is that it contains an empty **`if`**. Even more significant, it now contains an **`else`**, which means it is in violation of ***Never use `if` with `else`*** [R4.1.1]. A common effect of joining classes in the manner we just did is that it exposes potentially hidden type codes. In this case, the boolean `falling` is a type code. We can expose this type code by making it into an enum.

**Listing 5.26. Before**

```
/// ...
  new Stone(true);
/// ...
  new Stone(false);
/// ...
class Stone implements Tile {
  constructor(private falling:
      boolean) {}
  // ...
  isFallingStone() { return
      this.falling; }
}
```

**Listing 5.27. After**

```
enum FallingState {
  FALLING, RESTING
}
/// ...
  new Stone(FallingState.FALLING);
/// ...
  new Stone(FallingState.RESTING);
/// ...
class Stone implements Tile {
  constructor(private falling: FallingState) {}
  // ...
  isFallingStone() { return this.falling ===
      FallingState.FALLING; }
}
```

This change has already made the code more readable because we get away with the unnamed boolean arguments to `Stone`. But even better; we know how to deal with enums: ***Replace type code with classes*** [P4.1.3].

**Listing 5.28. Before**

```
enum FallingState {
  FALLING, RESTING
}
  new Stone(FallingState.FALLING);
  new Stone(FallingState.RESTING);
class Stone implements Tile {
  constructor(private falling: FallingState)
      {}
  // ...
  isFallingStone() { return
      this.falling.isFalling() ===
      FallingState.FALLING; }
}
```

**Listing 5.29. After**

```
interface FallingState {
  isFalling(): boolean;
  isResting(): boolean;
}
class Falling {
  isFalling() { return true; }
  isResting() { return false; }
}
class Resting {
  isFalling() { return false; }
  isResting() { return true; }
}
  new Stone(new Falling());
  new Stone(new Resting());
class Stone implements Tile {
  constructor(private falling:
      FallingState) {}
  // ...
  isFallingStone() { return
      this.falling.isFalling(); }
}
```

As mentioned earlier, if we are bothered that the **`new`**s are slightly slower, we can extract them to constants; but remember, performance optimization should be guided by profiling tools. If we inline `isFallingStone` in the method `moveHorizontal`, we see

that we should probably use ***Push code into classes*** [P4.1.5].

**Listing 5.30. Before**

```
interface FallingState {
  // ...
}
class Falling {
  // ...
}
class Resting {
  // ...
}
class Stone implements Tile {
  // ...
  moveHorizontal(dx: number) {
    if (!this.falling.isFalling()) {
      if (map[playery][playerx + dx +
      dx].isAir()
        && !map[playery + 1][playerx +
      dx].isAir()) {
        map[playery][playerx + dx + dx]
      = this;
        moveToTile(playerx + dx,
      playery);
      }
    } else if (this.falling.isFalling())
      {
    }
  }
}
```

**Listing 5.31. After**

```
interface FallingState {
  // ...
  moveHorizontal(tile: Tile, dx:
      number): void;
}
class Falling {
  // ...
  moveHorizontal(tile: Tile, dx: number)
      {
  }
}
class Resting {
  // ...
  moveHorizontal(tile: Tile, dx: number)
      {
    if (map[playery][playerx + dx +
      dx].isAir()
      && !map[playery + 1][playerx +
      dx].isAir()) {
      map[playery][playerx + dx + dx] =
      tile;
      moveToTile(playerx + dx, playery);
    }
  }
}
class Stone implements Tile {
  // ...
  moveHorizontal(dx: number) {
    this.falling.moveHorizontal(this,
      dx);
  }
}
```

Finally, since we introduced a new interface, we can use ***Try delete then compile*** [P4.5.1] to remove `isResting`. I leave it to you to do the same for `Box` and `FallingBox`; notice that you can reuse `FallingState`. We call unifying two similar classes like this ***Unify similar classes*** [P5.1.1].

### 5.1.1 *Refactoring:* Unify similar classes

#### DESCRIPTION

Whenever we have two or more classes that differ from each other only in a few constant methods, we can use this refactoring pattern to unify them. A set of constant methods is called a *basis*. A basis with two methods is called a *two-point basis*. We want our basis to have as few methods as possible. When we want to unify $X$ classes, we need at most an $(X - 1)$-point basis. Unifying classes is great because having fewer classes usually means we uncover more structure.

### PROCESS

1. The first phase is to make all the non-basis methods equal. For each of these methods, perform these steps:
   a. In the body of each version of the method, add an enclosing `if (true) { }` around the existing code.
   b. Replace `true` with an expression calling all the basis methods and comparing their result to their constant values.
   c. Copy the body of each version and paste it with an `else` into all the other versions.
2. Now that only the basis methods are different, the second phase begins by introducing a field for each method in the basis and assigning its constant in the constructor.
3. Change the methods to return the new fields instead of the constants.
4. Compile to ensure that we have not broken anything yet.
5. For each class, one field at a time:
   a. Copy the default value of the field, and then make the default value a parameter.
   b. Go through the compiler errors, and insert the default value as an argument.
6. After all the classes are identical, delete all but one of the unified classes, and fix all the compile errors by switching to the remaining class.

### EXAMPLE

In this example, we have three color classes with only one method, which returns a constant.

**Listing 5.32. Initial**

```
function string2color(str: string) {
  if (str === "red") return new Red();
  else if (str === "green") return new Green();
  else if (str === "blue") return new Blue();
}
class Red {
  hexCode() { return "#ff0000"; }
}
class Green {
  hexCode() { return "#00ff00"; }
}
class Blue {
  hexCode() { return "#0000ff"; }
}
```

We want to unify the three classes, so we follow the process:

1. In this case, there are no methods for phase one to unify.
2. Introduce a field for the methods that returns a different constant for each of them, and assign each method's constant in the constructor.

**Listing 5.33. Before**

```
class Red {
  hexCode() { return "#ff0000"; }
}
class Green {
  hexCode() { return "#00ff00"; }
}
class Blue {
  hexCode() { return "#0000ff"; }
}
```

**Listing 5.34. After (1/4)**

```
class Red {
  constructor(private hex: string =
      "#ff0000") {} (1)
  hexCode() { return "#ff0000"; }
}
class Green {
  constructor(private hex: string =
      "#00ff00") {} (1)
  hexCode() { return "#00ff00"; }
}
class Blue {
  constructor(private hex: string =
      "#0000ff") {} (1)
  hexCode() { return "#0000ff"; }
}
```

(1) Added constructors

3. Change the methods to return the new fields instead of the constants.

**Listing 5.35. Before**

```
class Red {
  // ...
  hexCode() { return "#ff0000"; }
}
class Green {
  // ...
  hexCode() { return "#00ff00"; }
}
class Blue {
  // ...
  hexCode() { return "#0000ff"; }
}
```

**Listing 5.36. After (2/4)**

```
class Red {
  // ...
  hexCode() { return this.hex; } (1)
}
class Green {
  // ...
  hexCode() { return this.hex; } (1)
}
class Blue {
  // ...
  hexCode() { return this.hex; } (1)
}
```

(1) Returning field instead

4. Compile to ensure that we have not broken anything yet.
5. For each class, one field at a time:
   a. Copy the default value of the field, and then make the default value a parameter.

**Listing 5.37. Before**

```
class Red {
  constructor(private hex:
      string = "#ff0000") {}
  // ...
}
```

**Listing 5.38. After (3/4)**

```
class Red {
  constructor(private hex: string) {} (1)
  // ...
}
```

(1) Removes the default value

   b. Go through the compiler errors, and insert the default value as an argument.

**Listing 5.39. Before**

```
function string2color(str: string) {
  if (str === "red")
    return new Red();
  else if (str === "green")
    return new Green();
  else if (str === "blue")
    return new Blue();
}
```

**Listing 5.40. After (4/4)**

```
function string2color(str: string) {
  if (str === "red")
    return new Red("#ff0000"); (1)
  else if (str === "green")
    return new Green();
  else if (str === "blue")
    return new Blue();
}
```

(1) Calling with the default value

6. After all the classes are identical, delete all but one of the unified classes, and fix all the compile errors by switching to the remaining class.

In the end, we have the following code.

**Listing 5.41. Before**

```
function string2color(str: string) {
  if (str === "red")
    return new Red();
  else if (str === "green")
    return new Green();
  else if (str === "blue")
    return new Blue();
}
class Red {
  hexCode() { return "#ff0000"; }
}
class Green {
  hexCode() { return "#00ff00"; }
}
class Blue {
  hexCode() { return "#0000ff"; }
}
```

**Listing 5.42. After**

```
function string2color(str: string) {
  if (str === "red")
    return new Color("#ff0000");
  else if (str === "green")
    return new Color("#00ff00");
  else if (str === "blue")
    return new Color("#0000ff");
}
class Color {
  constructor(private hex: string)
      {}
  hexCode() { return this.hex; }
}
```

At this point, it might make sense to extract the three colors into constants to avoid having to instantiate them over and over again. Luckily, this is trivial to do.

**Listing 5.43. Before**

```
function string2color(str: string) {
  if (str === "red")
    return new Color("#ff0000");
  else if (str === "green")
    return new Color("#00ff00");
  else if (str === "blue")
    return new Color("#0000ff");
}
```

**Listing 5.44. After**

```
function string2color(str: string) {
  if (str === "red")
    return RED; (1)
  else if (str === "green")
    return GREEN; (1)
  else if (str === "blue")
    return BLUE; (1)
}
const RED = new Color("#ff0000"); (2)
const GREEN = new Color("#00ff00"); (2)
const BLUE = new Color("#0000ff"); (2)
```

(1) Uses constants to avoid **new**

(2) The new constant definitions

**FURTHER READING**

To my knowledge, this is the first description of this process as a refactoring pattern.

## 5.2 Unifying simple conditions

To proceed with `updateTile`, we would like to make the bodies of some of the **`if`**s more similar. Let's look at the code.

**Listing 5.45. Initial**

```
function updateTile(x: number, y: number) {
  if (map[y][x].isStony() (1)
          && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Stone(new Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy() (1)
          && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone()) {
    map[y][x] = new Stone(new Resting());
  } else if (map[y][x].isFallingBox()) {
    map[y][x] = new Box(new Resting());
  }
}
```

(1) New method for setting the new field; empty in most classes

We decide to introduce methods for setting and unsetting the new `falling` field.

**Listing 5.46. After introducing `drop` and `rest`**

```
interface Tile {
  // ...
  drop(): void; (1)
  rest(): void; (2)
}
class Stone implements Tile {
```

```
  // ...
  drop() { this.falling = new Falling(); } (1)
  rest() { this.falling = new Resting(); } (2)
}
class Flux implements Tile {
  // ...
  drop() {} (1)
  rest() {} (2)
}
```

(1) New method for setting the new field; empty in most classes

(2) New method for unsetting the new field; empty in most classes

We can use `rest` directly in `updateTile`.

**Listing 5.47. Before**

```
function updateTile(x: number, y:
      number) {
  if (map[y][x].isStony()
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Stone(new
      Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new
      Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone())
      {
    map[y][x] = new Stone(new
      Resting());
  } else if (map[y][x].isFallingBox()) {
    map[y][x] = new Box(new Resting());
  }
}
```

**Listing 5.48. After**

```
function updateTile(x: number, y:
      number) {
  if (map[y][x].isStony()
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Stone(new
      Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new
      Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone())
      {
    map[y][x].rest(); (1)
  } else if (map[y][x].isFallingBox()) {
    map[y][x].rest(); (1)
  }
}
```

(1) Uses the new helper method

We see that the body of the last two **`if`**s is the same. When two **`if`** statements that are next to each other have the same body, we can join them by simply putting a `||` between the two conditions.

**Listing 5.49. Before**

```
function updateTile(x: number, y:
      number) {
  if (map[y][x].isStony()
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Stone(new
      Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new
      Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone())
      {
    map[y][x].rest();
  } else if (map[y][x].isFallingBox()) {
    map[y][x].rest();
  }
}
```

**Listing 5.50. After**

```
function updateTile(x: number, y:
      number) {
  if (map[y][x].isStony()
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Stone(new
      Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new
      Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone()
      (1)
         || map[y][x].isFallingBox()) {
      (1)
    map[y][x].rest();
  }
}
```

(1) Combined condition

We're used to ||s by now, so it should come as no surprise that we immediately push the || expression into the classes, naming them after what the two method names have in common: `isFalling`.

I want to repeat an important point from chapter 2. Throughout this process, we are not making any judgments: we are simply following the code's existing structure. We are doing these refactorings without really knowing what the code does. This is important, because refactoring can be expensive if you have to first understand all of the code. The fact that some refactoring patterns are possible without studying the code can save you considerable time.

The resulting code looks like this.

**Listing 5.51. Before**

```
function updateTile(x: number, y: number)
      {
  if (map[y][x].isStony()
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Stone(new
      Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new
      Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFallingStone()
         || map[y][x].isFallingBox()) {
    map[y][x].rest();
  }
}
```

**Listing 5.52. After**

```
function updateTile(x: number, y:
      number) {
  if (map[y][x].isStony()
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Stone(new
      Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
         && map[y + 1][x].isAir()) {
    map[y + 1][x] = new Box(new
      Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
      (1)
    map[y][x].rest();
  }
}
```

(1) Uses the new helper method

Even though this refactoring pattern is one of the simplest in the book, its power enables more powerful ones. Without further ado, here is ***Combine `if`s*** [P5.2.1].

## 5.2.1 *Refactoring:* Combine *`if`*s

### DESCRIPTION

This refactoring pattern reduces duplication by joining consecutive **`if`**s that have identical bodies. We usually encounter this condition only during targeted refactoring, where we deliberately try to make it happen — it is unnatural to write **`if`**s with identical bodies next to each other. This pattern is useful because it exposes a relation in the two expressions by adding an `||`, which — as we have seen — we like to take advantage of.

### PROCESS

1. Verify that the bodies are indeed the same.
2. Select the code between the closing parenthesis of the first **`if`** and the opening parenthesis of the **`else if`**, press Delete, and insert a `||`. Insert an opening parenthesis after the **`if`** and a closing parenthesis before `{`. We always keep the parentheses around the expressions to make sure we do not change the behavior.

**Listing 5.53. Before**

```
if (expression1) {
  // body
} else if (expression2) {
  // same body
}
```

**Listing 5.54. After**

```
if ((expression1) || (expression2)) {
  // body
}
```

3. If the expressions are simple, we can remove the superfluous parentheses or configure our editor to do it.

#### EXAMPLE

In this example, we have some logic to determine what to do with an invoice.

**Listing 5.55. Initial**

```
if (today.getDate() === 1 && account.getBalance() > invoice.getAmount()) {
  account.pay(bill);
} else if (invoice.isLastDayOfPayment() && invoice.isApproved()) {
  account.pay(bill);
}
```

We follow the process:

1. Verify that the bodies are indeed the same.
2. Select the code between the closing parenthesis of the first **`if`** and the opening parenthesis of the **`else if`**, press Delete, and insert a `||`. Insert an opening parenthesis after the **`if`** and a closing parenthesis before `{`. We always keep the parentheses around the expressions to make sure we do not change the behavior.

**Listing 5.56. Before**

```
if (today.getDate() === 1
    && account.getBalance() >
      invoice.getAmount()) {
  account.pay(bill);
} else if
      (invoice.isLastDayOfPayment()
    && invoice.isApproved()) {
  account.pay(bill);
}
```

**Listing 5.57. After**

```
if ((today.getDate() === 1 (1)
    && account.getBalance() >
      invoice.getAmount()) (1)
    || (invoice.isLastDayOfPayment()
      (2)
    && invoice.isApproved())) { (2)
  account.pay(bill);
}
```

(1) Condition of the first `if` (parenthesized)

(2) Condition of the second `if` (parenthesized)

3. If the expressions are simple, we can remove the superfluous parentheses or configure our editor to do it.

#### FURTHER READING

Many people in the industry consider this common knowledge. So, I think this is the first description of it as an official refactoring pattern.

## 5.3 *Unifying complex conditions*

Looking at the first **`if`** of `updateTile`, we realize that it simply replaces one stone with air, and one air with stone. This is the same as moving the stone tile and setting it to falling using the `drop` function. The same is true for the box case.

**Listing 5.58. Before**

```
function updateTile(x: number, y:
      number) {
  if (map[y][x].isStony()
        && map[y + 1][x].isAir())
      {
    map[y + 1][x] = new Stone(new
      Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
        && map[y + 1][x].isAir())
      {
    map[y + 1][x] = new Box(new
      Falling());
    map[y][x] = new Air();
  } else if (map[y][x].isFalling())
      {
    map[y][x].rest();
  }
}
```

**Listing 5.59. After**

```
function updateTile(x: number, y: number) {
  if (map[y][x].isStony()
        && map[y + 1][x].isAir()) {
    map[y][x].drop(); (1)
    map[y + 1][x] = map[y][x]; (1)
    map[y][x] = new Air(); (1)
  } else if (map[y][x].isBoxy()
        && map[y + 1][x].isAir()) {
    map[y][x].drop(); (1)
    map[y + 1][x] = map[y][x]; (1)
    map[y][x] = new Air(); (1)
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

(1) Sets the stone or box to fall, swaps the tiles, and puts in new air

Now we are in a position where the bodies of the two first **`if`**s are the same. We can again use ***Combine `if`s*** [P5.2.1] to join the two **`if`**s into a single **`if`** by putting a `||` between the conditions.

**Listing 5.60. Before**

```
function updateTile(x: number, y:
      number) {
  if (map[y][x].isStony()
        && map[y + 1][x].isAir())
      {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isBoxy()
        && map[y + 1][x].isAir())
      {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isFalling())
      {
    map[y][x].rest();
  }
}
```

**Listing 5.61. After**

```
function updateTile(x: number, y: number) {
  if (map[y][x].isStony() (1)
        && map[y + 1][x].isAir()) (1)
        || map[y][x].isBoxy() (1)
        && map[y + 1][x].isAir()) { (1)
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

(1) Combined conditions

The resulting condition is slightly more complex than last time. Therefore, this is a good time to discuss how to work with such conditions.

### 5.3.1 Arithmetic of conditions

We can manipulate conditional expressions the same way we do most of the code in this book: without knowing what it does. Without going into the theoretical background, it turns out that `||` (and `|`) behave like `+` (addition), and `&&` (and `&`) behave

like * (multiplication). This helps us remember when we need parentheses around ||, and all our regular arithmetic rules apply.

$$
\begin{aligned}
a + b + c = (a + b) + c = a + (b + c) &\qquad (+ \text{ is associative}) \\
a \cdot b \cdot c = (a \cdot b) \cdot c = a \cdot (b \cdot c) &\qquad (\cdot \text{ is associative}) \\
a + b = b + a &\qquad (+ \text{ is commutative}) \\
a \cdot b = b \cdot a &\qquad (\cdot \text{ is commutative}) \\
a \cdot (b + c) = a \cdot b + a \cdot c &\qquad (\cdot \text{ distributes over } + \text{ on the left}) \\
(a + b) \cdot c = a \cdot c + b \cdot c &\qquad (\cdot \text{ distributes over } + \text{ on the right})
\end{aligned}
$$

**Figure 5.1. Arithmetic rules**

The rules in figure 5.1 apply in all cases except when the conditions have side effects. To be able to use these rules as we expect, we should always avoid using side effects in conditions: ***Use pure conditions*** [R5.3.2].

### 5.3.2 *Rule:* Use pure conditions

#### STATEMENT

Conditions should always be pure.

#### EXPLANATION

*Conditions* are what come after **`if`** or **`while`** and what are in the middle part of **`for`** loops. *Pure* means the conditions do not have side effects. *Side effects* mean the conditions assign values to variables, throw exceptions, or interact with I/O, such as printing something, writing to files, etc.

Having pure conditions is important for multiple reasons. First, as mentioned, conditions with side effects prevent us from using the earlier rules. Second, side effects are uncommon in conditions, so we do not expect conditions to have side effects; this means it is something we need to discover, implying that we should spend more time investigating and more cognitive capacity keeping track of which conditions have which side-effects.

Code like the following is common, where `readLine` both returns the next line and advances the pointer. Advancing the pointer is a side effect, so our condition is not pure. A better implementation, on the right, separates the responsibility of getting the line and moving the pointer. It would be even better to also introduce a method that checks whether there is more to read, instead of returning `null`, but that is a discussion for another time.

**Listing 5.62. Before**

```
class Reader {
  private data: string[];
  private current: number;
  readLine() {
    return
      this.data[this.current++]
      || null;
  }
}
/// ...
let br = new Reader();
let line: string | null;
while ((line = br.readLine())
      !== null) {
  console.log(line);
}
```

**Listing 5.63. After**

```
class Reader {
  private data: string[];
  private current: number;
  nextLine() { (1)
    this.current++;
  }
  readLine() {
    return this.data[this.current] || null; (2)
  }
}
/// ...
let br = new Reader();
for (; br.readLine() !== null; br.nextLine()) {
      (3)
  let line = br.readLine(); (4)
  console.log(line);
}
```

(1) New method with a side effect
(2) Side effect removed from the existing method
(3) Changed to a `for` loop to ensure that we remember to call `nextLine`
(4) Second call to get the current line

Notice that we can call `readLine` as many times as we want to, with no side effects.

In cases where we do not have control over the implementation and therefore cannot split the return from the side effects, we can use a cache. There are many ways to implement caches; so, without going into detail about the implementation, here is a general-purpose cache that can take any method and split the side-effect part from the return part.

**Listing 5.64. Cacher**

```
class Cacher<T> {
  private cache: T;
  constructor(private mutator: () => T) {
    this.cache = this.mutator();
  }
  get() {
    return this.cache;
  }
  next() {
    this.cache = this.mutator();
  }
}

let tmpBr = new Reader(); (1)
let br = new Cacher(() => tmpBr.readLine()); (2)
for (; br.get() !== null; br.next()) {
  let line = br.get();
  console.log(line);
}
```

❶ Instantiating the `Reader` as usual, but with a temporary name
❷ Wraps the specific call in the cache

#### SMELL

This rule originates from a general smell that states, "Separate queries from commands"; you can find it in the book *Design by Contract, by Example* by Richard Mitchell and Jim McKim. For once, this smell is not difficult to get a feel for. In the smell, "commands" refers to anything with side effects, and "queries" means anything pure. An easy way to follow this smell is to only allow side effects in `void` methods: they either have side effects or return something, but not both.

The only difference between the general smell and this rule then is that we focus on the call site instead of the definition site. In the original work, Mitchell and McKim build more principles on top that rely on strict separation in all cases. We have loosened the smell to focus on conditions because mixing queries and commands outside conditions does not affect our ability to refactor; adhering to the the smell then is perhaps more a matter of style. It is also more common to have methods both return and mutate something, so we are practiced at spotting it. Indeed, one of the most common operators in programming, `++`, both increments and returns a value.

It is also easy to argue that rule also has roots in "a method should do only one thing" from Bob Martin's *Clean Code*. Having a side effect is one thing, and returning something is another.

#### INTENT

The intent is to separate getting data and changing data. This makes our code cleaner and more predictable. It usually also enables better naming, because the methods are simpler. Side effects fall under the category of mutating global state, which is dangerous, as described in chapter 2. Therefore, isolating the mutating makes it easier to manage.

#### REFERENCES

You can read about queries and commands and how to use them to make assertions — sometimes called *contracts* — in *Design by Contract, by Example* by Richard Mitchell and Jim McKim.

### 5.3.3 Using condition arithmetic

Working with conditions according to the rules in figure 5.1 is powerful. Consider our condition from `updateTile`: we first transform it into a math equation, after which we can easily use familiar arithmetic rules to simplify it and then transform it back into code. This transformation is illustrated in figure 5.2.

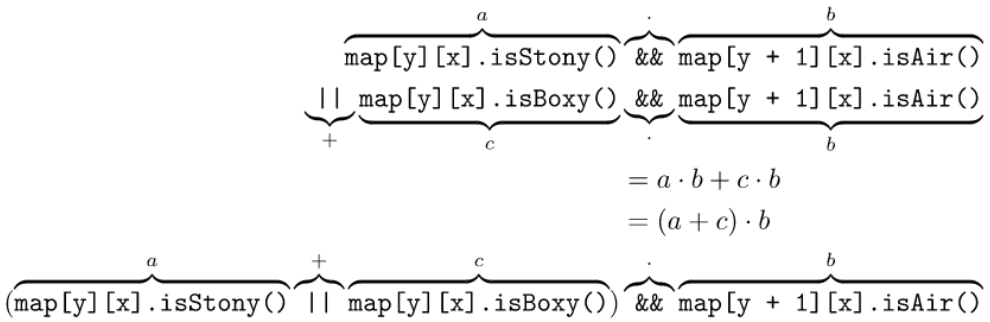


**Figure 5.2. Applying arithmetic rules**

Practicing the process of transforming a condition into a math equation, simplifying it, and changing it back to code in your head can be invaluable when you have to simplify more complex conditions in the real world. This technique can also help you spot tricky parenthesis errors in conditions.

### *A Story from Real Life*

I have spent so much time practicing this process that it is automatic to me. Several times in my career as a consultant, I have been brought onto a project for the sole purpose of tracking down errors with parentheses in conditions. If you haven't learned this trick, then these bugs are extremely difficult to spot, and their effects can seem unpredictable.

Putting our earlier simplification into the code, we get the following.

**Listing 5.65. Before**

```
function updateTile(x: number, y:
      number) {
  if (map[y][x].isStony()
         && map[y + 1][x].isAir())
         || map[y][x].isBoxy()
         && map[y + 1][x].isAir()) {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

**Listing 5.66. After**

```
function updateTile(x: number, y:
      number) {
  if ((map[y][x].isStony() (1)
         || map[y][x].isBoxy()) (1)
         && map[y + 1][x].isAir()) {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

(1) Condition simplified, with a parenthesis

Now we are in a situation similar to earlier: we have a `||` that we want to push into the classes. In chapter 4, we had a relation between stones and boxes and called the method `pushable`. However, that name does not make sense in this situation. It is important not to blindly reuse a name just because it addresses the same relation: it should also include the context. So, in this case we write a new method called `canFall`.

After ***Push code into classes*** [P4.1.5], we have another nice simplification.

**Listing 5.67. Before**

```
function updateTile(x: number, y:
      number) {
  if ((map[y][x].isStony()
         || map[y][x].isBoxy())
         && map[y + 1][x].isAir()) {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

**Listing 5.68. After**

```
function updateTile(x: number, y: number)
      {
  if (map[y][x].canFall() (1)
         && map[y + 1][x].isAir()) {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

(1) Uses the new helper method

## 5.4 Unifying code across classes

Continuing with `updateTile`, there is nothing more to postpone pushing it into classes.

**Listing 5.69. Before**

```
function updateTile(x: number, y:
      number) {
  if (map[y][x].canFall()
         && map[y + 1][x].isAir()) {
    map[y][x].drop();
    map[y + 1][x] = map[y][x];
    map[y][x] = new Air();
  } else if (map[y][x].isFalling()) {
    map[y][x].rest();
  }
}
```

**Listing 5.70. After**

```
function updateTile(x: number, y: number)
      {
  map[y][x].update(x, y);
}
interface Tile {
  // ...
  update(x: number, y: number): void;
}
class Air implements Tile {
  // ...
  update(x: number, y: number) { }
}
class Stone implements Tile {
  // ...
  update(x: number, y: number) {
    if (map[y + 1][x].isAir()) {
      this.falling = new Falling();
      map[y + 1][x] = this;
      map[y][x] = new Air();
    } else if (this.falling.isFalling()) {
      this.falling = new Resting();
    }
  }
}
```

We inline `updateTile` to clean up. Having pushed a lot of methods into our classes, we have introduced a lot of methods in our interface. This is a good time to do some midway cleaning with ***Try delete then compile*** [P4.5.1]. Notice that this removes almost all the `isX` methods we have introduced. The ones we are left with all have some sort of special meaning, like the locks.

Currently, we have this exact code in both `Stone` and `Box`. Contrary to the situation we had earlier (section 4.3), this is not a place where we want divergence. The falling behavior should stay in sync, and it also seems like something that we might use again

later if we introduce more tiles.

1. We first make a new `FallStrategy` class.

**Listing 5.71. New class**

```
class FallStrategy {
}
```

2. Instantiate `FallStrategy` in the constructor of `Stone` and `Box`.

**Listing 5.72. Before**

```
class Stone implements Tile {
  constructor(private falling:
      FallingState) {}
  // ...
}
```

**Listing 5.73. After (1/5)**

```
class Stone implements Tile {
  private fallStrategy: FallStrategy;
        (1)
  constructor(private falling:
      FallingState) {
    this.fallStrategy = new
      FallStrategy(); (2)
  }
  // ...
}
```

(1) New field

(2) Initializes the new field

3. We move `update` the same way we do with ***Push code into classes*** [P4.1.5].

**Listing 5.74. Before**

```
class Stone implements Tile {
  // ...
  update(x: number, y: number) {
    if (map[y + 1][x].isAir()) {
      this.falling = new Falling();
      map[y + 1][x] = this;
      map[y][x] = new Air();
    } else if
      (this.falling.isFalling()) {
      this.falling = new Resting();
    }
  }
}
class FallStrategy {
}
```

**Listing 5.75. After (2/5)**

```
class Stone implements Tile {
  update(x: number, y: number) {
    this.fallStrategy.update(x, y);
  }
}
class FallStrategy {
  update(x: number, y: number) {
    if (map[y + 1][x].isAir()) {
      this.falling = new Falling();
      map[y + 1][x] = this;
      map[y][x] = new Air();
    } else if
      (this.falling.isFalling()) {
      this.falling = new Resting();
    }
  }
}
```

4. We are dependent on the `falling` field, so we:
   a. Move the `falling` field, and make an accessor for it in `FallStrategy`.

**Listing 5.76. Before**

```
class Stone implements Tile {
  private fallStrategy:
      FallStrategy;
  constructor(private falling:
      FallingState) {
    this.fallStrategy = new
      FallStrategy();
  }
  // ...
}
class FallStrategy {
  // ...
}
```

**Listing 5.77. After (3/5)**

```
class Stone implements Tile {
  private fallStrategy: FallStrategy;
  constructor(falling: FallingState) { (1)
    this.fallStrategy = new
      FallStrategy(falling); (2)
  }
  // ...
}
class FallStrategy {
  constructor(private falling:
      FallingState) { } (3)
  getFalling() { return this.falling; } (4)
  // ...
}
```

(1) Removing `private`
(2) Add an argument
(3) Adding a constructor with a parameter
(4) New accessor for the field

b. Fix errors in the original class by using the new accessors.

**Listing 5.78. Before**

```
class Stone implements Tile {
  // ...
  moveHorizontal(dx: number) {

      this.falling.moveHorizon
      tal(this, dx);
  }
}
```

**Listing 5.79. After (4/5)**

```
class Stone implements Tile {
  // ...
  moveHorizontal(dx: number) {

      this.fallStrategy.getFalling().moveH
      orizontal(this, dx); (1)
  }
}
```

(1) Using the new accessor

5. Add a `tile` parameter to replace `this` for the remaining errors in `FallStrategy`.

**Listing 5.80. Before**

```
class Stone implements Tile {
  // ...
  update(x: number, y: number) {
    this.fallStrategy.update(x, y);
  }
}
class FallStrategy {
  update(x: number, y: number) {
    if (map[y + 1][x].isAir()) {
      this.falling = new Falling();
      map[y + 1][x] = this;
      map[y][x] = new Air();
    } else if (this.falling) {
      this.falling = new Resting();
    }
  }
}
```

**Listing 5.81. After (5/5)**

```
class Stone implements Tile {
  // ...
  update(x: number, y: number) {
    this.fallStrategy.update(this, x, y); (1)
  }
}
class FallStrategy {
  update(tile: Tile, x: number, y: number) {
      (1)
    if (map[y + 1][x].isAir()) {
      this.falling = new Falling();
      map[y + 1][x] = tile; (2)
      map[y][x] = new Air();
    } else if (this.falling) {
      this.falling = new Resting();
    }
  }
}
```

(1) Adding a parameter to replace `this`

This results in the following transformation.

**Listing 5.82. Before**

```
class Stone implements Tile {
  constructor(private falling:
      FallingState) {}
  // ...
  update(x: number, y: number) {
    if (map[y + 1][x].isAir()) {
      this.falling = new Falling();
      map[y + 1][x] = this;
      map[y][x] = new Air();
    } else if (this.falling) {
      this.falling = new Resting();
    }
  }
}
```

**Listing 5.83. After**

```
class Stone implements Tile {
  private fallStrategy: FallStrategy;
  constructor(falling: FallingState) {
    this.fallStrategy = new
      FallStrategy(falling);
  }
  // ...
  update(x: number, y: number) {
    this.fallStrategy.update(this, x, y);
  }
}
class FallStrategy {
  constructor(private falling: FallingState)
      { }
  isFalling() { return this.falling; }
  update(tile: Tile, x: number, y: number) {
    if (map[y + 1][x].isAir()) {
      this.falling = new Falling();
      map[y + 1][x] = tile;
      map[y][x] = new Air();
    } else if (this.falling) {
      this.falling = new Resting();
    }
  }
}
```

In `FallStrategy.update`, if we look closely at the **`else if`**, we see that if `falling` is **`true`**, it is set to **`false`**; otherwise, it is already **`false`**. So we can remove the condition.

**Listing 5.84. Before**

```
class FallStrategy {
  // ...
  update(tile: Tile, x: number, y:
      number) {
    if (map[y + 1][x].isAir()) {
      this.falling = new Falling();
      map[y + 1][x] = tile;
      map[y][x] = new Air();
    } else if (this.falling) {
      this.falling = new Resting();
    }
  }
}
```

**Listing 5.85. After**

```
class FallStrategy {
  // ...
  update(tile: Tile, x: number, y: number)
      {
    if (map[y + 1][x].isAir()) {
      this.falling = new Falling();
      map[y + 1][x] = tile;
      map[y][x] = new Air();
    } else { (1)
      this.falling = new Resting();
    }
  }
}
```

(1) Removed condition

Now the code assigns `falling` in all paths, so we can factor it out. We also remove the empty **`else`**. We then have an **`if`** that checks the same value as the variable; in such cases, we like to use the variable directly, instead.

**Listing 5.86. Before**

```
class FallStrategy {
  // ...
  update(tile: Tile, x: number, y:
      number) {
    if (map[y + 1][x].isAir()) {
      this.falling = new Falling();
      map[y + 1][x] = tile;
      map[y][x] = new Air();
    } else { (1)
      this.falling = new Resting();
    }
  }
}
```

**Listing 5.87. After**

```
class FallStrategy {
  // ...
  update(tile: Tile, x: number, y: number)
      {
    this.falling = map[y + 1][x].isAir() ?
      new Falling() : new Resting(); (1)
    if (this.falling.isFalling()) {
      map[y + 1][x] = tile;
      map[y][x] = new Air();
    }
  }
}
```

(1) Factoring `this.falling` out of **`if`**

We are within five lines! But we are not done yet. Remember that we have a rule stating ***`if` only at the start*** [R3.5.1]. We still need to follow this rule, so we do a simple ***Extract method*** [P3.2.1].

**Listing 5.88. Before**

```
class FallStrategy {
  // ...
  update(tile: Tile, x: number, y:
      number) {
    this.falling = map[y +
      1][x].isAir() ? new Falling()
      : new Resting();
    if (this.falling.isFalling()) {
      map[y + 1][x] = tile;
      map[y][x] = new Air();
    }
  }
}
```

**Listing 5.89. After**

```
class FallStrategy {
  // ...
  update(tile: Tile, x: number, y: number) {
    this.falling = map[y + 1][x].isAir() ?
      new Falling() : new Resting();
    this.drop(tile, x, y); (1)
  }
  private drop(tile: Tile, x: number, y:
      number){ (1)
    if (this.falling.isFalling()) {
      map[y + 1][x] = tile;
      map[y][x] = new Air();
    }
  }
}
```

(1) Extracted method

Inline `updateTile`, compile, test, commit, and take a break.

The refactoring pattern we went through to unify the "fall code" is called ***Introduce strategy-pattern*** [P5.4.2]. It is the most sophisticated refactoring pattern in this book. It is also referenced in many other places, all of which use diagrams to demonstrate its effect. We don't want to go against tradition, so we first need to take a detour for a primer in UML class diagrams.

### 5.4.1 UML Class Diagrams 101

Sometimes we need to communicate properties about code like its architecture or the order in which things happen. Some of these properties are easier to convey with diagrams; therefore we have a framework called Unified Modeling Language (UML).

UML comprises many types of standard diagrams to convey specific properties about code. A few examples include sequence diagrams, class diagrams, and activity diagrams. Explaining all of these is out of the scope of this book. The strategy pattern — and some other patterns — are most commonly demonstrated with a specific type of UML diagram called *class diagrams*. It is my goal that after you read this book, you will be able to pick up any other book about clean code or refactoring and understand it. So, this section explains how class diagrams work.

Class diagrams illustrate the structures of interfaces and classes and how they relate to each other. We represent classes with boxes, a title, and sometimes methods, but rarely fields. Interfaces are represented like classes but with **`interface`** above the title. We can also denote whether methods and fields are **`private`** (-) or **`public`** (+). Here is how a small class with fields and methods is depicted in a class diagram.

**Listing 5.90. A complete class**

```
class Cls {
  private text: string = "Hello";
  public name: string;
  private getText() { return
      this.text; }
  printText() {
      console.log(getText()); }
}
```

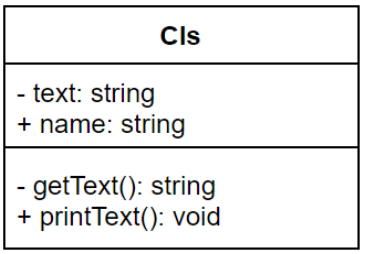


**Figure 5.3. Class diagram**

In most cases, it is only interesting to talk about the public interface of a class. Thus we usually don't include anything private. Most fields are private â€“ for good reason, as we discuss in the next chapter. Since we often depict only public methods, we don't need to include visibility.

The most important part of a class diagram is the relations between the classes and interfaces. They fall into three categories "X uses a Y," "X is a Y," and "X has a Y" or "X has Ys." Within each of these categories, two specific arrow types communicate slightly different things. The types of relations depicted in a class diagram are shown in figure 5.4.

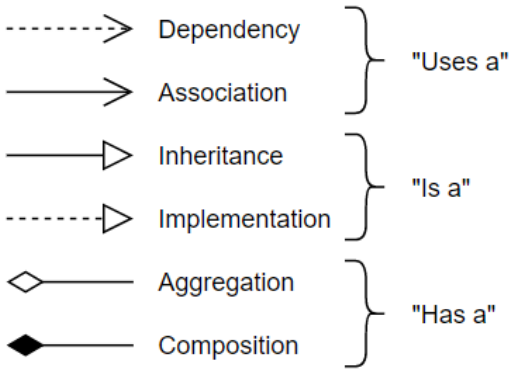


**Figure 5.4. UML relations**

We can simplify this a bit. The rule ***Only inherit from interfaces*** [R4.3.2] prevents us from using the inheritance arrow. The "uses" arrows are generally used when we don't know or don't care what the relation is. The difference between composition and aggregation is mostly aesthetic. So, most of the time, we can get away with two of the relation types: composition and implementation. Here are two simple uses of classes and diagrams.

**Listing 5.91. Implements**

```
interface A {
  m(): void;
}
class B implements A {
  m() { console.log("Hello"); }
}
```

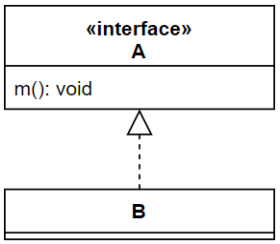


**Figure 5.5. Implementation**

Notice that we do not need to show that `B` also has an `m` method, because the interface already tells us that.

**Listing 5.92. Composed**

```
class A {
  private b: B;
}
class B {
}
```


**Figure 5.6. Composition**

Making class diagrams for an entire program quickly becomes overwhelming and thus is not helpful. We use them mostly to illustrate design patterns or small parts of the software architecture, so we only include important methods. Figure 5.7 shows a class diagram focusing on `FallStrategy`.

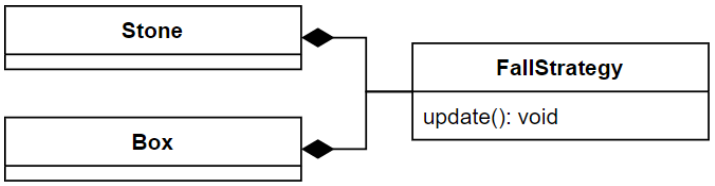


**Figure 5.7. Class diagram with a focus on `FallStrategy`**

Armed with the knowledge of how to use class diagrams, we can illustrate the effect of ***Introduce strategy-pattern*** [P5.4.2].

## 5.4.2 *Refactoring:* Introduce strategy-pattern

### DESCRIPTION

We have already discussed how an **`if`** statement is a low-level control flow operator. We have also mentioned how using objects is advantageous. The concept of introducing variance by instantiating another class is called the *strategy* pattern. It is commonly illustrated with a class diagram similar to the one above, see figure 5.3.

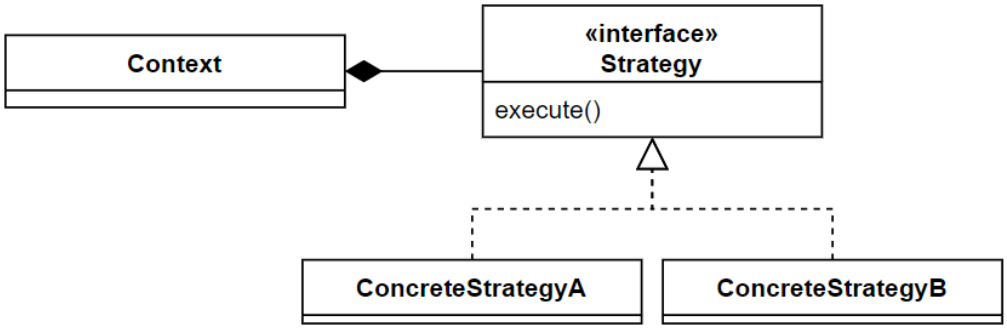


**Figure 5.8. Strategy pattern as a class diagram**

Many patterns are variations of the strategy pattern; if our strategy has fields, we call it a state pattern, instead. These distinctions are mostly academic — they make us sound smart, but in practice, knowing the correct names does not add much to our communication. The fundamental idea is the same: enable change by adding classes (we discussed the advantages of doing this in chapter 2). For this reason, we use the term *strategy pattern* to describe moving any code into its own class. When we do not use the new variation option, we have still added the possibility.

Notice that this is different from transforming type codes into classes. Those classes represent data, and as such, we tend to push lots of methods into them. We rarely add methods to strategy classes after they are finished; instead, we prefer to create a new class if we need to change functionality.

Because variance is the purpose of the strategy pattern, it is always depicted with inheritance: usually from an interface, but sometimes from an abstract class. We have already discussed the disadvantages of that, but we did not use inheritance.

The variance of the strategy pattern is the ultimate form of late binding. At runtime, the strategy pattern allows us to load classes that are completely unknown to our code and to seamlessly integrate them into our control flow. No need to even recompile the code. If you take only one thing away from this book, let it be how powerful and useful the strategy pattern is.

There are two situations for introducing a strategy pattern. First, we can refactor because we want to introduce variation in the code. In this case, we should have an interface in the end. However, to make this refactoring as quick as possible, we recommend postponing the interface. Second, in the situation with the fall code, we do not expect to add variance any time soon; we merely wish to unify behavior across classes. We have a rule stating ***No interface with only one implementation*** [R5.4.3]. When we need the interface — whether immediately or later — we use a refactoring pattern called ***Extract interface from implementation*** [P5.4.4]. Both the rule and the refactoring are explained next.

### PROCESS

1. Perform ***Extract method*** [P3.2.1] on the code we want to isolate. If we want to

unify it with something else, make sure the methods are identical.
2. Make a new class.
3. Instantiate the new class in the constructor.
4. Move the method into the new class.
5. If there are dependencies on any fields:
   a. Move along any fields to the new class, making accessors for the fields.
   b. Fix errors in the original class by using the new accessors.
6. Add a parameter to replace `this` for the remaining errors in the new class.
7. ***Inline method*** [P4.1.7] to reverse the extraction from step 1.

#### EXAMPLE

Say in our application calculates prices in two different places, both of which add taxes. This is a place where we might want variation, but it is also something we should unify.

**Listing 5.93. Initial**

```
class Store {
  calculatePrice(products: Product[]) {
    // ...
    let total = subtotal * 1.25;
    return total;
  }
}
class Online {
  calculatePrice(products: Product[]) {
    // ...
    let total = subtotal * 1.25;
    return total;
  }
}
```

We follow the process:

1. Perform ***Extract method*** [P3.2.1] on the code we want to isolate. If we want to unify it with something else, make sure the methods are identical.

**Listing 5.94. Before**

```
class Store {
  calculatePrice(products: Product[])
      {
    // ...
    let total = subtotal * 1.25;
    return total;
  }
}
class Online {
  calculatePrice(products: Product[])
      {
    // ...
    let total = subtotal * 1.25;
    return total;
  }
}
```

**Listing 5.95. After (1/4)**

```
class Store {
  calculatePrice(products: Product[])
      {
    // ...
    let total = withTaxes(subtotal);
      (1)
    return total;
  }
  withTaxes(subtotal: number) { (1)
    return subtotal * 1.25;
  }
}
class Online {
  calculatePrice(products: Product[])
      {
    // ...
    let total = withTaxes(subtotal);
      (1)
    return total;
  }
  withTaxes(subtotal: number) { (1)
    return subtotal * 1.25;
  }
}
```

(1) Extracted method and call

2. Make a new class.

**Listing 5.96. After**

```
class Tax {
}
```

3. Instantiate the new class in the constructor.

**Listing 5.97. Before**

```
class Store {
  // ...
}
class Online {
  // ...
}
```

**Listing 5.98. After (2/4)**

```
class Store {
  private taxes: Tax; (1)
  constructor() { (1)
    taxes = new Tax(); (1)
  }
  // ...
}
class Online {
  private taxes: Tax; (1)
  constructor() { (1)
    taxes = new Tax(); (1)
  }
  // ...
}
```

(1) Adding a field and initializing it in the constructor

4. Move the method into `Tax`.

**Listing 5.99. Before**

```
class Store {
  // ...
  withTaxes(subtotal: number)
      {
    return subtotal * 1.25;
  }
}
class Online {
  // ...
  withTaxes(subtotal: number){
    return subtotal * 1.25;
  }
}
class Tax {
}
```

**Listing 5.100. After (3/4)**

```
class Store {
  // ...
  withTaxes(subtotal: number) {
    return this.taxes.withTaxes(subtotal);
      (2)
  }
}
class Online {
  // ...
  withTaxes(subtotal: number){
    return this.taxes.withTaxes(subtotal);
      (2)
  }
}
class Tax {
  withTaxes(subtotal: number) { (1)
    return subtotal * 1.25;
  }
}
```

(1) New method

(2) Calling the method in the class

5. In this case, there are no dependencies on any fields, so we skip these steps:
   a. Move along any fields to the `Tax` class, making accessors for fields.
   b. Fix errors in the original class by using the new accessors.
6. Add a parameter to replace `this` for the remaining errors in the new class. This is unnecessary in this case since there are no errors in the new class.
7. ***Inline method*** [P4.1.7] to reverse the extraction from step 1.

**Listing 5.101. Before**

```
class Store {
  // ...
  calculatePrice(products: Product[]) {
    // ...
    let total = withTaxes(subtotal);
    return total;
  }
  withTaxes(subtotal: number) {
    return
      this.taxes.withTaxes(subtotal);
  }
}
class Online {
  // ...
  calculatePrice(products: Product[]) {
    // ...
    let total = withTaxes(subtotal);
    return total;
  }
  withTaxes(subtotal: number){
    return
      this.taxes.withTaxes(subtotal);
  }
}
```

**Listing 5.102. After (4/4)**

```
class Store {
  // ...
  calculatePrice(products: Product[]) {
    // ...
    let total =
      this.taxes.withTaxes(subtotal);
    return total;
  }
  (1)
}
class Online {
  // ...
  calculatePrice(products: Product[]) {
    // ...
    let total =
      this.taxes.withTaxes(subtotal);
    return total;
  }
  (1)
}
```

(1) `withTaxes` is removed.

#### FURTHER READING

The strategy pattern was first introduced in *Design Patterns* by the Gang of Four: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Because it is so powerful, it can be found many places. However, the idea of post-imposing the strategy pattern into code comes from Martin Fowler's book *Refactoring*.

### 5.4.3 *Rule:* No interface with only one implementation

#### STATEMENT

Never have interfaces with only one implementation.

#### EXPLANATION

This rule states that we should not have interfaces with a single implementation. These lonesome interfaces often come from learning advice such as "Always code up against an interface." However, this approach is not always beneficial.

A simple argument is that an interface with only one implementation does not add readability. Even worse, an interface signals variation; and if there is none, it adds overhead to our mental model. It may also slow us down if we want to modify the implementing class, as we also need to update the interface, which we have to be more careful with. The argument is similar to that of ***Specialize method*** [P4.2.2]; interfaces with only one implementing class are a form of generalization that is not helpful.

In many languages, we place interfaces in their own file. In such languages, having an

interface with one implementing class uses two files, whereas having only the implementing class uses only one. A difference of one file is not a significant issue; but if our codebase have an affinity for interfaces with only one descendant, we may have twice as many files as we should, which incurs a major mental overhead.

There are cases where it makes sense to have interfaces with zero implementations. These are useful when we want to make anonymous classes, most commonly for things such as comparators, or to enforce stricter encapsulation through anonymous inner classes. We discuss encapsulation in the next chapter; however, since anonymous inner classes are rarely used in practice, they are out of the scope of this book.

#### SMELL

A famous saying states, "Every problem in computer science can be solved by introducing another layer of indirection." This is exactly what interfaces are. We hide details under an abstraction. John Carmack was the brilliant lead programmer on Doom, Quake, and several other games. This rule originates from a smell he explicited in one of his tweets: "Abstraction trades an increase in real complexity **for** a decrease in perceived complexity" — implying that we should be careful with our abstractions.

#### INTENT

The intention is to limit unnecessary boilerplate code. Interfaces are a common source of boilerplate; they are especially dangerous because lots of people have been taught that interfaces are always preferable, so they tend to bloat their applications.

#### REFERENCES

Fred George presented a similar rule in his 2015 GOTO talk "The Secret Assumption of Agile."

### 5.4.4 *Refactoring:* Extract interface from implementation

#### DESCRIPTION

This is another rather simple refactoring. It is useful since it allows us to postpone making interfaces until they are needed (when we want to introduce variance).

#### PROCESS

1. Create a new interface with the same name as the class we are extracting from.
2. Rename the class we want to extract the interface from, and make it implement the new interface.
3. Compile, and go through the errors:
   a. If the error is caused by a `new`, change it to the new class name.
   b. Otherwise, add the method that is causing the error to the interface.

#### EXAMPLE

Let's continue with the earlier example.

**Listing 5.103. Initial**

```
class Store {
  private taxes: Tax;
  constructor() {
    taxes = new Tax();
  }
  calculatePrice(products: Product[]) {
    let subtotal = sumPrices(products);
    let total = this.taxes.withTaxes(subtotal);
    return total;
  }
}
class Tax {
  withTaxes(subtotal: number) {
    return subtotal * 1.25;
  }
}
```

We follow the process:

1. Create a new interface with the same name as the class we are extracting from.

**Listing 5.104. Adding new interface**

```
interface Tax {
}
```

2. Rename the class we want to extract the interface from, and make it implement the new interface.

**Listing 5.105. Before**

```
class Tax {
  // ...
}
```

**Listing 5.106. After (1/3)**

```
class DanishTax implements Tax {
  // ...
}
```

3. Compile, and go through the errors:
   a. If it is a **`new`**, change it to the new class name.

**Listing 5.107. Before**

```
class Store {
  private taxes: Tax;
  constructor() {
    taxes = new Tax();
  }
  // ...
}
```

**Listing 5.108. After (2/3)**

```
class Store {
  private taxes: Tax;
  constructor() {
    taxes = new DanishTax(); (1)
  }
  // ...
}
```

(1) Instantiates a class instead of an interface

   b. Otherwise, add the method that is causing the error to the interface.

**Listing 5.109. Before**

```
class Store {
  // ...
  calculatePrice(products: Product[])
      {
    let subtotal =
      sumPrices(products);
    let total =
      this.taxes.withTaxes(subtotal)
      ;
    return total;
  }
}
interface Tax {
}
```

**Listing 5.110. After (3/3)**

```
class Store {
  // ...
  calculatePrice(products: Product[])
      {
    let subtotal =
      sumPrices(products);
    let total =
      this.taxes.withTaxes(subtotal)
      ;
    return total;
  }
}
interface Tax {
  withTaxes(subtotal: number):
      number; (1)
}
```

(1) Adding the method to the interface

**FURTHER READING**

To our knowledge, this is the first description of this technique as a refactoring pattern.

## 5.5 Unifying similar functions

Another place we have similar code is in the two functions `removeLock1` and `removeLock2`.

**Listing 5.111. `removeLock1`**

```
function removeLock1() {
  for (let y = 0; y < map.length;
      y++) {
    for (let x = 0; x <
      map[y].length; x++) {
      if (map[y][x].isLock1()) { (1)
        map[y][x] = new Air();
      }
    }
  }
}
```

**Listing 5.112. `removeLock2`**

```
function removeLock2() {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length; x++) {
      if (map[y][x].isLock2()) { (1)
        map[y][x] = new Air();
      }
    }
  }
}
```

(1) The only difference

As it turns out, we can use ***Introduce strategy-pattern*** [P5.4.2] to unify these as well. They are not identical, so we handle them by pretending we have the first one and need to introduce the second one: that is, we want to add variance.

1. Start by performing ***Extract method*** [P3.2.1] on the code we want to isolate.

**Listing 5.113. Before**

```
function removeLock1() {
  for (let y = 0; y < map.length;
       y++) {
    for (let x = 0; x <
      map[y].length; x++) {
      if (map[y][x].isLock1()) {
        map[y][x] = new Air();
      }
    }
  }
}
```

**Listing 5.114. After (1/3)**

```
function removeLock1() {
  for (let y = 0; y < map.length;
       y++) {
    for (let x = 0; x <
      map[y].length; x++) {
      if (check(map[y][x])) { (1)
        map[y][x] = new Air();
      }
    }
  }
}
function check(tile: Tile) { (1)
  return tile.isLock1();
}
```

(1) New method and call

2. Make a new class.

**Listing 5.115. A new class**

```
class RemoveStrategy {
}
```

3. In this case, we have no constructor where we can instantiate this new class. Instead, we instantiate it directly in the function.

**Listing 5.116. Before**

```
function removeLock1() {
  for (let y = 0; y < map.length;
       y++) {
    for (let x = 0; x <
      map[y].length; x++) {
      if (check(map[y][x])) {
        map[y][x] = new Air();
      }
    }
  }
}
```

**Listing 5.117. After (2/3)**

```
function removeLock1() {
  let shouldRemove = new
      RemoveStrategy(); (1)
  for (let y = 0; y < map.length;
       y++) {
    for (let x = 0; x <
      map[y].length; x++) {
      if (check(map[y][x])) {
        map[y][x] = new Air();
      }
    }
  }
}
```

(1) Initializing new class

4. Move the method.

**Listing 5.118. Before**

```
function removeLock1() {
  let shouldRemove = new
      RemoveStrategy();
  for (let y = 0; y < map.length;
      y++) {
    for (let x = 0; x <
      map[y].length; x++) {
      if (check(map[y][x])) {
        map[y][x] = new Air();
      }
    }
  }
}
function check(tile: Tile) {
  return tile.isLock1();
}
```

**Listing 5.119. After (3/3)**

```
function removeLock1() {
  let shouldRemove = new
      RemoveStrategy();
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length;
      x++) {
      if (shouldRemove.check(map[y][x]))
      { (1)
        map[y][x] = new Air();
      }
    }
  }
}
class RemoveStrategy {
  check(tile: Tile) { return
      tile.isLock1(); } (1)
}
```

(1) Moved method

5. There are no dependencies on any fields and no errors in the new class.

Having introduced a strategy, we can use ***Extract interface from implementation*** [P5.4.4] in preparation to introduce the variance:

1. Create a new interface with the same name as the class we are extracting from.

**Listing 5.120. Before**

```
interface RemoveStrategy {
}
```

2. Rename the class we want to extract the interface from, and make it implement the new interface.

**Listing 5.121. Before**

```
class RemoveStrategy {
  // ...
}
```

**Listing 5.122. After (1/3)**

```
class RemoveLock1 implements RemoveStrategy {
  // ...
}
```

3. Compile, and go through the errors:
   a. If it is a **`new`**, change it to the new class name.

**Listing 5.123. Before**

```
function removeLock1() {
  let shouldRemove = new
      RemoveStrategy();
  for (let y = 0; y < map.length;
      y++) {
    for (let x = 0; x <
      map[y].length; x++) {
      if
      (shouldRemove.check(map[y][x])
      ) {
        map[y][x] = new Air();
      }
    }
  }
}
```

**Listing 5.124. After (2/3)**

```
function removeLock1() {
  let shouldRemove = new
      RemoveLock1(); (1)
  for (let y = 0; y < map.length;
      y++) {
    for (let x = 0; x <
      map[y].length; x++) {
      if
      (shouldRemove.check(map[y][x])
      ) {
        map[y][x] = new Air();
      }
    }
  }
}
```

(1) Instantiating a class instead of an interface

b. Otherwise, add the method that is causing the error to the interface.

**Listing 5.125. Before**

```
interface RemoveStrategy {
}
```

**Listing 5.126. After (3/3)**

```
interface RemoveStrategy {
  check(tile: Tile): boolean;
}
```

At this point, it is trivial to make `RemoveLock2` from a copy of `RemoveLock1`. We then only need to move `shouldRemove` out as a parameter. I'll spare you the details, but we do the following:

1. Extracting from `removeLock1` everything but the first line we get `remove`.
2. The local variable `shouldRemove` is only used once, so we inline it.
3. ***Inline method*** [P4.1.7] on `removeLock1`.

These refactorings result in us having only one `remove`.

**Listing 5.127. Before**

```
function removeLock1() {
  for (let y = 0; y < map.length; y++)
      {
    for (let x = 0; x < map[y].length;
      x++) {
      if (map[y][x].isLock1()) {
        map[y][x] = new Air();
      }
    }
  }
}
class Key1 implements Tile {
  // ...
  moveHorizontal(dx: number) {
    removeLock1();
    moveToTile(playerx + dx, playery);
  }
}
```

**Listing 5.128. After**

```
function remove(shouldRemove:
      RemoveStrategy) {
  for (let y = 0; y < map.length; y++) {
    for (let x = 0; x < map[y].length;
      x++) {
      if (shouldRemove.check(map[y][x])) {
        map[y][x] = new Air();
      }
    }
  }
}
class Key1 implements Tile {
  // ...
  moveHorizontal(dx: number) {
    remove(new RemoveLock1());
    moveToTile(playerx + dx, playery);
  }
}
interface RemoveStrategy {
  check(tile: Tile): boolean;
}
class RemoveLock1 implements
      RemoveStrategy {
  check(tile: Tile) { return
      tile.isLock1(); }
}
```

Just like earlier, this makes `remove` more general, but this time without limiting us. It also enables change by addition: if we want to remove another type of tile, we can simply make another class that implements `RemoveStrategy` without modifying anything.

In some applications, we like to avoid calling **`new`** inside a loop. If that is the case here, then we can easily store the `RemoveLock` strategy in an instance variable and initialize it in the constructor. However, we are not finished with `Key1` yet.

## 5.6 Unifying similar code

We also have some duplication in `Key1` and `Key2`, and `Lock1` and `Lock2`. In each case, the twin classes are almost identical.

**Listing 5.129. Key1 and Lock1**

```
class Key1 implements Tile {
  // ...
  draw(g: CanvasRenderingContext2D, x:
      number, y: number) {
    g.fillStyle = "#ffcc00";
    g.fillRect(x * TILE_SIZE, y *
      TILE_SIZE, TILE_SIZE, TILE_SIZE);
  }
  moveHorizontal(dx: number) {
    remove(new RemoveLock1());
    moveToTile(playerx + dx, playery);
  }
}
class Lock1 implements Tile {
  // ...
  isLock1() { return true; }
  isLock2() { return false; }
  draw(g: CanvasRenderingContext2D, x:
      number, y: number) {
    g.fillStyle = "#ffcc00";
    g.fillRect(x * TILE_SIZE, y *
      TILE_SIZE, TILE_SIZE, TILE_SIZE);
  }
}
```

**Listing 5.130. Key2 and Lock2**

```
class Key2 implements Tile {
  // ...
  draw(g: CanvasRenderingContext2D, x:
      number, y: number) {
    g.fillStyle = "#00ccff";
    g.fillRect(x * TILE_SIZE, y *
      TILE_SIZE, TILE_SIZE, TILE_SIZE);
  }
  moveHorizontal(dx: number) {
    remove(new RemoveLock2());
    moveToTile(playerx + dx, playery);
  }
}
class Lock2 implements Tile {
  // ...
  isLock1() { return false; }
  isLock2() { return true; }
  draw(g: CanvasRenderingContext2D, x:
      number, y: number) {
    g.fillStyle = "#00ccff";
    g.fillRect(x * TILE_SIZE, y *
      TILE_SIZE, TILE_SIZE, TILE_SIZE);
  }
}
```

We first use ***Unify similar classes*** [P5.1.1] on both locks and both keys.

**Listing 5.131. Before**

```
class Key1 implements Tile {
  // ...
  draw(g: CanvasRenderingContext2D, x:
      number, y: number) {
    g.fillStyle = "#ffcc00";
    g.fillRect(x * TILE_SIZE, y *
      TILE_SIZE, TILE_SIZE, TILE_SIZE);
  }
  moveHorizontal(dx: number) {
    remove(new RemoveLock1());
    moveToTile(playerx + dx, playery);
  }
}
class Lock1 implements Tile {
  // ...
  draw(g: CanvasRenderingContext2D, x:
      number, y: number) {
    g.fillStyle = "#ffcc00";
    g.fillRect(x * TILE_SIZE, y *
      TILE_SIZE, TILE_SIZE, TILE_SIZE);
  }
}
function transformTile(tile: RawTile) {
  switch (tile) {
    // ...
    case RawTile.KEY1: return new
      Key1();
    case RawTile.LOCK1: return new
      Lock1();
  }
}
```

**Listing 5.132. After**

```
class Key implements Tile {
  constructor(
    private color: string,
    private removeStrategy:
      RemoveStrategy) { }
  // ...
  draw(g: CanvasRenderingContext2D, x:
      number, y: number) {
    g.fillStyle = this.color;
    g.fillRect(x * TILE_SIZE, y *
      TILE_SIZE, TILE_SIZE, TILE_SIZE);
  }
  moveHorizontal(dx: number) {
    remove(this.removeStrategy);
    moveToTile(playerx + dx, playery);
  }
}
class Lock implements Tile {
  constructor(
    private color: string,
    private lock1: boolean,
    private lock2: boolean) { }
  // ...
  isLock1() { return this.lock1; }
  isLock2() { return this.lock2; }
  draw(g: CanvasRenderingContext2D, x:
      number, y: number) {
    g.fillStyle = this.color;
    g.fillRect(x * TILE_SIZE, y *
      TILE_SIZE, TILE_SIZE, TILE_SIZE);
  }
}
function transformTile(tile: RawTile) {
  switch (tile) {
    // ...
    case RawTile.KEY1: return new
      Key("#ffcc00", new RemoveLock1());
    case RawTile.LOCK1: return new
      Lock("#ffcc00", true, false);
  }
}
```

This code works, but we can take advantage of some structure that we already know. We introduced the methods `isLock1` and `isLock2`: they came from two values in an enum, so we know that only one if these methods can return **`true`** for the any given class. We therefore need only one parameter to represent both methods. The same is true for the `Lock` methods.

**Listing 5.133. Before**

```
class Lock implements Tile {
  constructor(
    private color: string,
    private lock1: boolean,
    private lock2: boolean) { }
  // ...
  isLock1() { return this.lock1; }
  isLock2() { return this.lock2; }
}
```

**Listing 5.134. After**

```
class Lock implements Tile {
  constructor(
    private color: string,
    private lock1: boolean) { }
  // ...
  isLock1() { return this.lock1; }
  isLock2() { return !this.lock1; }
}
```

It also seems as though there is a connection between the parameters `color`, `lock1`, and `removeStrategy` of our constructors in `Key` and `Lock`. When we want to unify things across two classes, we use our favorite new trick: ***Introduce strategy-pattern*** [P5.4.2].

**Listing 5.135. Key1 and Lock1**

```
class Key implements Tile {
  constructor(
    private color: string,
    private removeStrategy: RemoveStrategy) { }
  // ...
  draw(g: CanvasRenderingContext2D, x: number, y:
      number) {
    g.fillStyle = this.color;
    g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
      TILE_SIZE, TILE_SIZE);
  }
  moveHorizontal(dx: number) {
    remove(this.removeStrategy);
    moveToTile(playerx + dx, playery);
  }
  moveVertical(dy: number) {
    remove(this.removeStrategy);
    moveToTile(playerx, playery + dy);
  }
}
class Lock implements Tile {
  constructor(
    private color: string,
    private lock1: boolean) { }
  // ...
  isLock1() { return this.lock1; }
  isLock2() { return !this.lock1; }
  draw(g: CanvasRenderingContext2D, x: number, y:
      number) {
    g.fillStyle = this.color;
    g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
      TILE_SIZE, TILE_SIZE);
  }
}
function transformTile(tile: RawTile) {
  switch (tile) {
    // ...
    case RawTile.KEY1: return new Key("#ffcc00", new
      RemoveLock1());
    case RawTile.LOCK1: return new Lock("#ffcc00",
      true);
  }
}
```

**Listing 5.136. After *Introduce strategy-pattern* [P5.4.2]**

```
class Key implements Tile {
  constructor(private keyConf: KeyConfiguration) { }
  // ...
  draw(g: CanvasRenderingContext2D, x: number, y:
      number) {
    g.fillStyle = this.keyConf.getColor();
    g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
      TILE_SIZE, TILE_SIZE);
  }
  moveHorizontal(dx: number) {
    remove(this.keyConf.getRemoveStrategy());
    moveToTile(playerx + dx, playery);
  }
  moveVertical(dy: number) {
    remove(this.keyConf.getRemoveStrategy());
    moveToTile(playerx, playery + dy);
  }
}
class Lock implements Tile {
  constructor(private keyConf: KeyConfiguration) { }
  // ...
  isLock1() { return this.keyConf.is1(); }
  isLock2() { return !this.keyConf.is1(); }
  draw(g: CanvasRenderingContext2D, x: number, y:
      number) {
    g.fillStyle = this.keyConf.getColor();
    g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
      TILE_SIZE, TILE_SIZE);
  }
}
class KeyConfiguration {
  constructor(private color: string, private _1:
      boolean, private removeStrategy:
      RemoveStrategy) { }
  getColor() { return this.color; }
  is1() { return this._1; }
  getRemoveStrategy() { return this.removeStrategy; }
}
const YELLOW_KEY = new KeyConfiguration("#ffcc00",
      true, new RemoveLock1());
function transformTile(tile: RawTile) {
  switch (tile) {
    // ...
    case RawTile.KEY1: return new Key(YELLOW_KEY);
    case RawTile.LOCK1: return new Lock(YELLOW_KEY);
  }
}
```

Imagine that at this point we want to introduce a third and fourth key+lock pair. We do this by changing the `boolean` in `keyConfiguration` to a `number` and changing the `isLock` methods to a single `fits(id: number)`. We can now introduce as many key+locks as we want. Of course, after this, we rewrite the `number` to an enum and then use ***Replace type code with classes*** [P4.1.3] — and you know the rest.

Again, note that this transformation made explicit something we have not spent time

investigating: the colors and the lock IDs are connected. We might have expected this due to the intuitive nature of the example. However, even if we were working on a complex financial system, we would slowly discover connections like these embedded in the code's existing structure. Some connections discovered this way are coincidental, so we have to be careful and ask ourselves whether this grouping makes sense. Such a grouping could also expose some nasty bugs in the code stemming from things being linked that are not supposed to be linked.

The `KeyConfiguration` class we introduced is currently pretty bare and boring. In the next chapter, we remedy this and we further expose and exploit links by encapsulating data.

## 5.7 Summary

- When we have similar code that should converge, we should unify it. We can unify classes with ***Unify similar classes*** [P5.1.1], **`if`**s with ***Combine `if`s*** [P5.2.1], and methods with ***Introduce strategy-pattern*** [P5.4.2].
- The rule ***Use pure conditions*** [R5.3.2] states that conditions should not have side effects, because if the do not then we can use conditional arithmetic. We saw how to use a `Cacher` to separate side effects from conditions.
- UML class diagrams are commonly used to illustrate specific architectural changes to a codebase.
- Interfaces with a single implementing class are a form of unnecessary generality. The rule ***No interface with only one implementation*** [R5.4.3] states that we should not have these. Instead, we should introduce the interface later with the refactoring pattern ***Extract interface from implementation*** [P5.4.4].

# 6 Encapsulate, to Localize Invariants

***This chapter covers:***

- Enforcing encapsulation with ***Do not use getters and setters*** [R6.1.1]
- Eliminating getters with ***Eliminate getter or setter*** [P6.1.3]
- Using ***Encapsulate data*** [P6.2.3] to ***Never have common affixes*** [R6.2.1]
- Eliminating an invariant with ***Enforce sequence*** [P6.4.1]

In chapter 2, we discussed the advantage of localizing invariants. We have already done that when introducing classes because they pull together functionality concerning the same data, and thereby also pulling the invariants closer and localizing them. In this chapter, we focus on encapsulation — limiting access to data and functionality — such that invariants can only be broken locally and therefore are much easier to prevent.

## 6.1 Encapsulating without getters

At this point, the code follows our rules and is already much more readable and extendable. However, we can do even better by introducing another rule: ***Do not use getters and setters*** [R6.1.1].

### 6.1.1 *Rule:* Do not use getters and setters

#### STATEMENT

Do not use setters or getters for non-boolean fields.

### EXPLANATION

When we say *setters and getters*, we mean methods that directly assign or return a non-boolean field, respectively. We also include C#'s properties. Notice that this has nothing to do with a method's name — it may or may not be called `getX`.

Getters and setters are often taught right alongside encapsulation as a standard method for getting around private fields. However, if we have getters for object fields, we immediately break encapsulation, and we are making our invariant global. After we return an object, the receiver can further distribute it, which we have no control over. Anyone who gets the object can call its public methods, possibly modifying it in a way we did not expect.

Setters present a similar issue. In theory, setters introduce another layer of indirection where we can change our internal data structure and modify our setter so it still has the same signature. Following our definition, such methods are no longer setters and thus are not a problem. However, what happens in practice is that we modify the setter to return the new data structure. Then the receiver has to be modified to accommodate this new data structure. This is exactly the form of tight coupling we want to avoid.

This is only a problem with mutable objects; however, the rule only specifies booleans as an exception. This is due to another effect of private fields that also applies to immutable fields: the architecture they suggest. One of the biggest advantages of making fields private is that doing so encourages a push-based architecture. In a push-based architecture, we push computations as close to the data as possible, whereas in a pull-based architecture, we fetch data and then do computations at a central point.

A pull-based architecture leads to a lot of "dumb" data classes without any interesting methods, and some big "manager" classes doing all the work and mixing data from a lot of places. This imposes a tight coupling between the data and the managers and, implicitly, between the data classes as well.

In a push-based architecture, instead of "getting" data, we pass data as arguments. As a result, all of our classes have functionality, and the code is distributed according to its utility.

In this example, we want to generate a link to a blog post. Both sides do the same thing, but one is written with a pull-based architecture and the other with a push-based architecture.

**Listing 6.1. Pull-based architecture**

```
class Website {
  constructor (private url: string) {}
  getUrl() { return this.url; }
}
class User {
  constructor (private username:
      string) {}
  getUsername() { return
      this.username; }
}
class BlogPost {
  constructor (private author: User,
      private id: string) {}
  getId() { return this.id; }
  getAuthor() { return this.author; }
}
function generatePostLink(website:
      Website, post: BlogPost) {
  let url = website.getUrl();
  let user = post.getAuthor();
  let name = user.getUsername();
  let postId = post.getId();
  return url + name + id;
}
```

**Listing 6.2. Push-based architecture**

```
class Website {
  constructor (private url: string) {}
  generateLink(name: string, id: string) {
      return this.url + name + id); }
}
class User {
  constructor (private username: string)
      {}
  generateLink(website: Website, id:
      string) { return
      website.generateLink(this.username,
      id); }
}
class BlogPost {
  constructor (private author: User,
      private id: string) {}
  generateLink(website: Website) { return
      this.author.generateLink(website,
      this.id); }
}
function generatePostLink(website:
      Website, post: BlogPost) {
  return post.generateLink(website);
}
```

In the push-based example, we would most likely inline `generatePostLink` as it is just a single line with no added information.

### SMELL

This rule is derived from something called the *Law of Demeter* that is often summarized as "Don't talk to strangers". A stranger in this context is an object that we do not have direct access to but can obtain a reference to. In object-oriented languages, this happens most commonly through getters — and therefore we have this rule.

### INTENT

The issue with interacting with objects to which we can obtain a reference is that we are now tightly coupled to the way we get the object. We know something about the internal structure of the owner of the object. The owner of the field cannot change the data structure without still supporting a way to get the old data structure; otherwise, it breaks our code.

In a push-based architecture, we expose methods like services. The users of those methods should not care about the internal structure of how we deliver them.

### REFERENCES

The Law of Demeter is described extensively online. For a thorough exercise that uses it, I recommend the Fantasy Battle refactoring kata by Samuel Ytterbrink, available at https://github.com/Neppord/FantasyBattle-Refactoring-Kata.

### 6.1.2 Applying it

In our code, we have only three getters, and two of them are in `KeyConfiguration`: `getColor` and `getRemoveStrategy`. Luckily they are not too difficult to deal with. We start with `getRemoveStrategy`.

1. Make `getRemoveStrategy` private to get errors everywhere we use it.

**Listing 6.3. Before**

```
class KeyConfiguration {
  // ...
  getRemoveStrategy() { return
      this.removeStrategy; }
}
```

**Listing 6.4. After (1/3)**

```
class KeyConfiguration {
  // ...
  private getRemoveStrategy() { return
      this.removeStrategy; } (1)
}
```

(1) Method made private

2. To fix the errors, use ***Push code into classes*** [P4.1.5] on the failing lines.

**Listing 6.5. Before**

```
class Key implements Tile {
  // ...
  moveHorizontal(dx: number) {

      remove(this.keyConf.getRemov
      eStrategy());
    moveToTile(playerx + dx,
      playery);
  }
  moveVertical(dy: number) {

      remove(this.keyConf.getRemov
      eStrategy());
    moveToTile(playerx, playery +
      dy);
  }
}
class KeyConfiguration {
  // ...
}
```

**Listing 6.6. After (2/3)**

```
class Key implements Tile {
  // ...
  moveHorizontal(dx: number) {
    this.keyConf.removeLock(); (1)
    moveToTile(playerx + dx, playery);
  }
  moveVertical(dy: number) {
    this.keyConf.removeLock(); (1)
    moveToTile(playerx, playery + dy);
  }
}
class KeyConfiguration {
  // ...
  removeLock() { (2)
    remove(this.removeStrategy);
  }
}
```

(1) Previously failing lines

(2) New method

3. `getRemoveStrategy` is inlined as part of ***Push code into classes*** [P4.1.5]. It is therefore unused, and we can delete it to avoid other people trying to use it.

**Listing 6.7. Before**

```
class KeyConfiguration {
  // ...
  private getRemoveStrategy() {
      return this.removeStrategy;
      }
}
```

**Listing 6.8. After (3/3)**

```
class KeyConfiguration {
  // ...
  (1)
}
```

(1) `getRemoveStrategy` is deleted.

After repeating this process for `getColor`, we have the following.

**Listing 6.9. Before**

```
class KeyConfiguration {
  // ...
  getColor() { return this.color; }
  getRemoveStrategy() { return this.removeStrategy; }
}
class Key implements Tile {
  // ...
  draw(g: CanvasRenderingContext2D, x: number, y:
      number) {
    g.fillStyle = this.keyConf.getColor();
    g.fillRect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE,
      TILE_SIZE);
  }
  moveHorizontal(dx: number) {
    remove(this.keyConf.getRemoveStrategy());
    moveToTile(playerx + dx, playery);
  }
  moveVertical(dy: number) {
    remove(this.keyConf.getRemoveStrategy());
    moveToTile(playerx, playery + dy);
  }
}
class Lock implements Tile {
  // ...
  draw(g: CanvasRenderingContext2D, x: number, y:
      number) {
    g.fillStyle = this.keyConf.getColor();
    g.fillRect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE,
      TILE_SIZE);
  }
}
```

**Listing 6.10. After**

```
class KeyConfiguration {
  // ...
  setColor(g: CanvasRenderingContext2D) { (1)
    g.fillStyle = this.color;
  }
  removeLock() { (2)
    remove(this.removeStrategy);
  }
}
class Key implements Tile {
  // ...
  draw(g: CanvasRenderingContext2D, x: number, y:
      number) {
    this.keyConf.setColor(g); (1)
    g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
      TILE_SIZE, TILE_SIZE);
  }
  moveHorizontal(dx: number) {
    this.keyConf.removeLock(); (2)
    moveToTile(playerx + dx, playery);
  }
  moveVertical(dy: number) {
    this.keyConf.removeLock(); (2)
    moveToTile(playerx, playery + dy);
  }
}
class Lock implements Tile {
  // ...
  draw(g: CanvasRenderingContext2D, x: number, y:
      number) {
    this.keyConf.setColor(g); (1)
    g.fillRect(x * TILE_SIZE, y * TILE_SIZE,
      TILE_SIZE, TILE_SIZE);
  }
}
```

(1) Method that replaces `getColor`

(2) Method that replaces `getRemoveStrategy`

Notice that `setColor` is not a setter in the sense described earlier.

Even though this is another simple process, having two names that suggest we should get rid of getters helps accentuate the importance of doing so. We call this refactoring pattern ***Eliminate getter or setter*** [P6.1.3].

### 6.1.3 *Refactoring:* Eliminate getter or setter

#### DESCRIPTION

This refactoring lets us eliminate getters and setters by moving the functionality closer to the data. Conveniently, because getters and setters are so similar, the same process can eliminate either, but for ease of reading we assume getters for the remainder of the description.

We have localized invariants many times by pushing code closer to the data. That is also the solution here. Usually, when we do so, we introduce a lot of similar functions instead of the getter. These are introduced based on how many contexts the getter is used in. Having many methods means we can name them based on the specific call context instead of the data context.

We saw an example of this issue in chapter 4. In the `TrafficLight` example, the car has a public method called `drive` that `TrafficLight` ends up calling. The `drive` method is named for the effect it has on the car, but we could instead name it based on the context it is called in: `notifyGreenLight`. The effect on the car is the same.

**Listing 6.11. Before**

```
class Green implements TrafficLight {
  // ...
  updateCar() { car.drive(); }
}
```

**Listing 6.12. After**

```
class Green implements TrafficLight {
  // ...
  updateCar() { car.notifyGreenLight(); }
      (1)
}
```

(1) After renaming the method based on the context

### PROCESS

4. Make the getter or setter private to get errors everywhere it is used.
5. Fix the errors with ***Push code into classes*** [P4.1.5].
6. The getter or setter is inlined as part of ***Push code into classes*** [P4.1.5]. It is therefore unused, so delete it to avoid other people trying to use it.

### EXAMPLE

Continuing the previous example, we can pick any getter to eliminate.

**Listing 6.13. Initial**

```
class Website {
  constructor (private url: string) {}
  getUrl() { return this.url; }
}
class User {
  constructor (private username: string) {}
  getUsername() { return this.username; }
}
class BlogPost {
  constructor (private author: User, private id: string) {}
  getId() { return this.id; }
  getAuthor() { return this.author; }
}
function generatePostLink(website: Website, post: BlogPost) {
  let url = website.getUrl();
  let user = post.getAuthor();
  let name = user.getUsername();
  let postId = post.getId();
  return url + name + id;
}
```

Here we demonstrate eliminating `getAuthor`. We follow the process:

1. Make the getter private to get errors everywhere it is used.

**Listing 6.14. Before**

```
class BlogPost {
  // ...
  getAuthor() { return this.author;
      }
}
```

**Listing 6.15. After (1/3)**

```
class BlogPost {
  // ...
  private getAuthor() { return
      this.author; } (1)
}
```

(1) Added private

2. Fix the errors with ***Push code into classes*** [P4.1.5].

**Listing 6.16. Before**

```
function generatePostLink(website:
      Website, post: BlogPost) {
  let url = website.getUrl();
  let user = post.getAuthor();
  let name = user.getUsername();
  let postId = post.getId();
  return url + name + id;
}
class BlogPost {
  // ...
}
```

**Listing 6.17. After (2/3)**

```
function generatePostLink(website:
      Website, post: BlogPost) {
  let url = website.getUrl();
  let name = post.getAuthorName();
  let postId = post.getId();
  return url + name + id;
}
class BlogPost {
  // ...
  getAuthorName() { return
      this.author.getUsername(); } (1)
}
```

(1) New method

3. The getter is inlined as part of ***Push code into classes*** [P4.1.5]. IT is therefore unused, so we delete to it avoid other people trying to use it.

**Listing 6.18. Before**

```
class BlogPost {
  // ...
  private getAuthor() { return
      this.author; }
}
```

**Listing 6.19. After (3/3)**

```
class BlogPost {
  // ...
  (1)
}
```

(1) `getAuthor` is deleted.

Following the same process for the other getters results in the push-based version described in section 6.1.1.

### 6.1.4 The final getter

The final getter is `FallStrategy.getFalling`. We follow the same process to get rid of it:

1. Make the getter private to get errors everywhere it is used.

**Listing 6.20. Before**

```
class FallStrategy {
  // ...
  getFalling() { return this.falling; }
}
```

**Listing 6.21. After (1/3)**

```
class FallStrategy {
  // ...
  private getFalling() { return
      this.falling; } (1)
}
```

(1) Added private

2. Fix the errors with ***Push code into classes*** [P4.1.5].

**Listing 6.22. Before**

```
class Stone implements Tile {
  // ...
  moveHorizontal(dx: number) {

      this.fallStrategy.getFalling().moveH
      orizontal(this, dx);
  }
}
class Box implements Tile {
  // ...
  moveHorizontal(dx: number) {

      this.fallStrategy.getFalling().moveH
      orizontal(this, dx);
  }
}
class FallStrategy {
  // ...
}
```

**Listing 6.23. After (2/3)**

```
class Stone implements Tile {
  // ...
  moveHorizontal(dx: number) {

      this.fallStrategy.moveHor
      izontal(this, dx); (1)
  }
}
class Box implements Tile {
  // ...
  moveHorizontal(dx: number) {

      this.fallStrategy.moveHor
      izontal(this, dx); (1)
  }
}
class FallStrategy {
  // ...
  moveHorizontal(tile: Tile,
      dx: number) {

      this.falling.moveHorizont
      al(tile, dx); (1)
  }
}
```

(1) New method

3. The getter is inlined as part of ***Push code into classes*** [P4.1.5]. It is therefore unused, so we delete it to avoid other people trying to use it.

**Listing 6.24. Before**

```
class FallStrategy {
  // ...
  private getFalling() { return
      this.falling; }
}
```

**Listing 6.25. After (3/3)**

```
class FallStrategy {
  // ...
  (1)
}
```

(1) `getFalling` is deleted.

This results in `FallStrategy` looking as follows.

**Listing 6.26. Before**

```
class Stone implements Tile {
  // ...
  moveHorizontal(dx: number) {

      this.fallStrategy.getFalling().moveHorizontal(t
      his, dx);
  }
}
class Box implements Tile {
  // ...
  moveHorizontal(dx: number) {

      this.fallStrategy.getFalling().moveHorizontal(t
      his, dx);
  }
}
class FallStrategy {
  constructor(private falling: FallingState) { }
  getFalling() { return this.falling; }
  update(tile: Tile, x: number, y: number) {
    this.falling = map[y + 1][x].isAir() ? new
      Falling() : new Resting();
    this.drop(tile, x, y);
  }
  private drop(tile: Tile, x: number, y: number) {
    if (this.falling.isFalling()) {
      map[y + 1][x] = tile;
      map[y][x] = new Air();
    }
  }
}
```

**Listing 6.27. After**

```
class Stone implements Tile {
  // ...
  moveHorizontal(dx: number) {
    this.fallStrategy.moveHorizontal(this, dx); (2)
  }
}
class Box implements Tile {
  // ...
  moveHorizontal(dx: number) {
    this.fallStrategy.moveHorizontal(this, dx); (2)
  }
}
class FallStrategy {
  constructor(private falling: FallingState) { }
  (1)
  update(tile: Tile, x: number, y: number) {
    this.falling = map[y + 1][x].isAir() ? new
      Falling() : new Resting();
    this.drop(tile, x, y);
  }
  private drop(tile: Tile, x: number, y: number) {
    if (this.falling.isFalling()) {
      map[y + 1][x] = tile;
      map[y][x] = new Air();
    }
  }
  moveHorizontal(tile: Tile, dx: number) { (2)
    this.falling.moveHorizontal(tile, dx);
  }
}
```

(1) `getFalling` is deleted

(2) New pushed code

Looking at `FallStrategy` we realize that we can make a few other improvements. First, the ternary operator `?  :` violates ***Never use `if` with `else`*** [R4.1.1]. Second, the **`if`** in `drop` seems to be more concerned more with `falling`. If we start with the ternary, we can get rid of it by pushing the line into `Tile`.

**Listing 6.28. Before**

```
interface Tile {
  // ...
}
class Air implements Tile {
  // ...
}
class Stone implements Tile {
  // ...
}
class FallStrategy {
  // ...
  update(tile: Tile, x: number, y: number) {
    this.falling = map[y + 1][x].isAir() ? new
      Falling() : new Resting();
    this.drop(tile, x, y);
  }
}
```

**Listing 6.29. After**

```
interface Tile {
  // ...
  getBlockOnTopState(): FallingState; (1)
}
class Air implements Tile {
  // ...
  getBlockOnTopState() { return new Falling(); } (1)
}
class Stone implements Tile {
  // ...
  getBlockOnTopState() { return new Resting(); } (1)
}
class FallStrategy {
  // ...
  update(tile: Tile, x: number, y: number) {
    this.falling = map[y +
      1][x].getBlockOnTopState(); (1)
    this.drop(tile, x, y);
  }
}
```

(1) Pushed code

In `FallStrategy.drop`, we can get rid of the **`if`** entirely by pushing the method into `FallingState` and inlining `FallStrategy.drop`.

**Listing 6.30. Before**

```
interface FallingState {
  // ...
}
class Falling {
  // ...
}
class Resting {
  // ...
}
class FallStrategy {
  // ...
  update(tile: Tile, x: number, y: number) {
    this.falling = map[y + 1][x].getBlockOnTopState();
    this.drop(tile, x, y);
  }
  private drop(tile: Tile, x: number, y: number) {
    if (this.falling.isFalling()) {
      map[y + 1][x] = tile;
      map[y][x] = new Air();
    }
  }
}
```

**Listing 6.31. After**

```
interface FallingState {
  // ...
  drop(tile: Tile, x: number, y: number): void; (1)
}
class Falling {
  // ...
  drop(tile: Tile, x: number, y: number) { (1)
    map[y + 1][x] = tile;
    map[y][x] = new Air();
  }
}
class Resting {
  // ...
  drop(tile: Tile, x: number, y: number) { } (1)
}
class FallStrategy {
  // ...
  update(tile: Tile, x: number, y: number) {
    this.falling = map[y +
      1][x].getBlockOnTopState();
    this.falling.drop(tile, x, y) (1)
  }
}
```

(1) Pushed code

## 6.2 Encapsulating simple data

Once again, we are in a position where our code abides by all of our rules. So, we

again introduce a new rule.

### 6.2.1 *Rule:* Never have common affixes

#### STATEMENT

Our code should not have methods or variables with common prefixes or suffixes.

#### EXPLANATION

We often postfix or prefix methods and variables with something that hints at their context, such as `username` for the name of the user or `startTimer` for a timer's start action. We do this to communicate the context. Although doing so makes the code more readable, when multiple elements have the same affix, it indicates coherence of these elements. There is a better way to communicate such structure: classes.

The advantage of using classes to group such methods and variables is that we have complete control over the external interface. We can hide helper methods so they do not pollute our global scope. This is especially valuable since our five-line rule introduces a lot of methods.

It can also be the case that not every method can be safely called from everywhere. If we extract the middle part of a complicated computation, it may require some setup before it works. In our game, this is the case for `updateMap` and `drawMap`, both of which require that `transformMap` has been called.

Most important, by hiding the data, we ensure that its invariants are maintained in the class. Doing so makes them local invariants, which are easier to maintain.

Consider the bank example from chapter 4, where we could deposit money without withdrawing it if we called `deposit` directly. Since we never want to call `deposit` directly, a better way to implement this functionality is to put both methods in a class and make `deposit` private.

**Listing 6.32. Bad**

```
function accountDeposit(to: string, amount: number) {
  let accountId = database.find(to);
  database.updateOne(accountId, { $inc: { balance:
      amount } });
}

function accountTransfer(from: string, to: string,
      amount: number) {
  accountDeposit(from, -amount);
  accountDeposit(to, amount);
}
```

**Listing 6.33. Good**

```
class Account {
  private deposit(to: string, amount: number) {
    let accountId = database.find(to);
    database.updateOne(accountId, { $inc: {
      balance: amount } });
  }

  transfer(from: string, to: string, amount:
      number) {
    this.deposit(from, -amount);
    this.deposit(to, amount);
  }
}
```

#### SMELL

The smell that this is derived from is called the *single responsibility principle*. It is the

same as the “Methods should do one thing” smell that we discussed earlier, but for classes. Classes should have a single responsibility.

#### INTENT

Designing classes with a single responsibility requires discipline and overview. This rule helps to identify sub-responsibilities. The structure hinted at by a common affix suggests that those methods and variables share the responsibility of the common affix; therefore, those methods should be in a separate class dedicated to this common responsibility.

This rule also helps us identify responsibilities even when they emerge over time as our application evolves. Classes often grow over time.

#### REFERENCES

The single responsibility principle is covered extensively on the internet. It is a standard design principle for classes. Unfortunately, this means it is often presented as something to design up front. But here we take a different approach and focus on a symptom that can be seen in the code.

### 6.2.2 *Applying it*

We have a clear group with the same affix, the method and variables:

- `playerx`
- `playery`
- `drawPlayer`

This suggests that we should put these in a class called `Player`. We already have a `Player` class, but it has a completely different purpose. There are two easy solutions. One is to enclose all tile types in a namespace and make them public. Although this is our preferred solution, it leads to a lot of TypeScript-specific tinkering. So, we choose the other easy solution and simply rename the existing `Player`.

**Listing 6.34. Before**

```
class Player implements Tile {
      /* ... */ }
```

**Listing 6.35. After**

```
class PlayerTile implements Tile { /* ... */ }
      (1)
```

(1) Append `Tile` to the name.

We can now make a new `Player` class for the group mentioned earlier.

1. Create a `Player` class.

   **Listing 6.36. New class**

   ```
   class Player { }
   ```

2. Move the variables `playerx` and `playery` into `Player`, replacing **`let`** with **`private`**. Remove `player` from their names. Also make

getters and setters for the variables, which will be dealt with later.

**Listing 6.37. Before**

```
let playerx = 1;
let playery = 1;
```

**Listing 6.38. After (1/4)**

```
class Player { (1)
  private x = 1; (2)
  private y = 1; (2)
  getX() { return this.x; } (3)
  getY() { return this.y; } (3)
  setX(x: number) { this.x = x; } (3)
  setY(y: number) { this.y = y; } (3)
}
```

(1) New class

(2) Remove `player` from the names.

(3) New getters and setters

3. Because `playerx` and `playery` are no longer in the global scope, the compiler helps us find all the references, by giving errors. We fix these errors in the following five steps:
   a. Pick a good variable name for an instance of the `Player` class: that is, `player`.
   b. Pretending that we have a `player` variable, use its getters or setters.

**Listing 6.39. Before**

```
function moveToTile(newx: number,
      newy: number) {
  map[playery][playerx] = new
      Air();
  map[newy][newx] = new
      PlayerTile();
  playerx = newx;
  playery = newy;
}
/// ...
```

**Listing 6.40. After (2/4)**

```
function moveToTile(newx: number, newy:
      number) {
  map[player.getY()][player.getX()] =
      new Air(); (1)
  map[newy][newx] = new PlayerTile();
  player.setX(newx); (2)
  player.setY(newy); (2)
}
/// ... (3)
```

(1) Access changed to getters

(2) Assignment changed to setters

(3) Access and assignment are changed everywhere.

   c. If we have errors in two or more different methods, we add `player: Player` as the first parameter and add `player` as the argument, causing new errors.

**Listing 6.41. Before**

```
interface Tile {
  // ...
  moveHorizontal(dx:
      number): void;
  moveVertical(dy: number):
      void;
}
```

**Listing 6.42. After (3/4)**

```
interface Tile {
  // ...
  moveHorizontal(player: Player, dx: number):
      void; (1)
  moveVertical(player: Player, dy: number):
      void; (1)
}
```

(1) `player` is added as a parameter to many methods, even those in the interfaces.

   d. Repeat until only one method errors.

e. Because we encapsulated variables, put `let player = new Player();` at the point where the variables used to be.

**Listing 6.43. After**

```
let player = new Player();
```

This transformation made changes throughout the codebase. The following are some of the important effects.

**Listing 6.44. Before**

```
interface Tile {
  // ...
  moveHorizontal(dx: number): void;
  moveVertical(dy: number): void;
}
/// ...
function moveToTile(newx: number, newy: number) {
  map[playery][playerx] = new Air();
  map[newy][newx] = new PlayerTile();
  playerx = newx;
  playery = newy;
}
/// ...
let playerx = 1;
let playery = 1;
```

**Listing 6.45. After (4/4)**

```
interface Tile {
  // ...
  moveHorizontal(player: Player, dx: number): void; (1)
  moveVertical(player: Player, dy: number): void;   (1)
}
/// ...
function moveToTile(newx: number, newy: number) {
  map[player.getY()][player.getX()] = new Air();    (4)
  map[newy][newx] = new PlayerTile();
  player.setX(newx); (5)
  player.setY(newy); (5)
}
/// ...
class Player {   (2)
  private x = 1; (2)
  private y = 1; (2)
  getX() { return this.x; } (2)
  getY() { return this.y; } (2)
  setX(x: number) { this.x = x; } (2)
  setY(y: number) { this.y = y; } (2)
}
let player = new Player();  (3)
```

(1) Added `player` as a parameter to lots of methods

(2) New class with getters and setters

(3) New declaration in place of the encapsulated variables

(4) Access changed to getters

(5) Assignments switched to setters

Having introduced a class, we can now push any method with a `Player` affix into this class without any issues. In this case, we only need to push `drawPlayer` into the class.

**Listing 6.46. Before**

```
function drawPlayer(player: Player, g:
      CanvasRenderingContext2D) {
  g.fillStyle = "#ff0000";
  g.fillRect(player.getX() * TILE_SIZE,
      player.getY() * TILE_SIZE,
      TILE_SIZE, TILE_SIZE);
}
class Player {
  // ...
}
```

**Listing 6.47. After**

```
function drawPlayer(player: Player, g:
      CanvasRenderingContext2D) {
  player.draw(g);
}
class Player {
  // ...
  draw(g: CanvasRenderingContext2D) {
    g.fillStyle = "#ff0000";
    g.fillRect(this.x * TILE_SIZE,
      this.y * TILE_SIZE, TILE_SIZE,
      TILE_SIZE); (1)
  }
}
```

(1) Notice that we have inlined the getters.

As usual, we perform ***Inline method*** [P4.1.7] on `drawPlayer`. The new class violates our new rule, ***Do not use getters and setters*** [R6.1.1]. So we use its related refactoring, ***Eliminate getter or setter*** [P6.1.3]. We start with `getX`.

1. Make the getter private to get errors everywhere it is used.

**Listing 6.48. Before**

```
class Player {
  // ...
  getX() { return this.x; }
}
```

**Listing 6.49. After (1/3)**

```
class Player {
  // ...
  private getX() { return this.x; } (1)
}
```

(1) Make the getter private.

2. Fix the errors with ***Push code into classes*** [P4.1.5].

**Listing 6.50. Before**

```
class Right implements Input {
  handle(player: Player) {
    map[player.getY()][player.getX() +
      1].moveHorizontal(player, 1);
  }
}
class Resting {
  // ...
  moveHorizontal(player: Player, tile: Tile, dx:
      number) {
    if (map[player.getY()][player.getX() + dx +
      dx].isAir()
      && !map[player.getY() + 1][player.getX() +
      dx].isAir()) {
      map[player.getY()][player.getX() + dx + dx] =
      tile;
      player.moveToTile(player.getX() + dx,
      player.getY());
    }
  }
}
/// ...
    player.moveToTile(player.getX(), player.getY() +
      dy);
/// ...
function moveToTile(player: Player, newx: number,
      newy: number) {
  map[player.getY()][player.getX()] = new Air();
  map[newy][newx] = new PlayerTile();
  player.setX(newx);
  player.setY(newy);
}
/// ...
class Player {
  // ...
}
```

**Listing 6.51. After (2/3)**

```
class Right implements Input {
  handle(player: Player) {
    player.moveHorizontal(1); (1)
  }
}
class Resting {
  // ...
  moveHorizontal(player: Player, tile: Tile, dx:
      number) {
    player.pushHorizontal(tile, dx); (1)
  }
}
/// ...
    player.move(0, dy); (1)
/// ...
function moveToTile(player: Player, newx: number, newy:
      number) {
  player.moveToTile(newx, newy);
}
/// ...
class Player {
  // ...
  moveHorizontal(dx: number) {
    map[this.y][this.x + dx].moveHorizontal(this, dx);
  }
  move(dx: number, dy: number) {
    moveToTile(this.x + dx, this.y + dy);
  }
  pushHorizontal(tile: Tile, dx: number) {
    if (map[this.y][this.x + dx + dx].isAir()
      && !map[this.y + 1][this.x + dx].isAir()) {
      map[this.y][this.x + dx + dx] = tile;
      moveToTile(this.x + dx, this.y);
    }
  }
  moveToTile(newx: number, newy: number) {
    map[this.y][this.x] = new Air();
    map[newy][newx] = new PlayerTile();
    this.x = newx;
    this.y = newy;
  }
}
```

(1) Methods pushed into `Player`

3. The getter is inlined as part of ***Push code into classes*** [P4.1.5]. It is therefore unused, so delete it to avoid other people trying to use it.

**Listing 6.52. Before**

```
class Player {
  // ...
  getX() { return this.x; }
}
```

**Listing 6.53. After (3/3)**

```
class Player {
  // ...
  (1)
}
```

(1) Delete `getX`

Luckily, `getX` and `getY` were so closely connected that `getY` simply disappeared with `getX`, along with (amazingly) the two setters. We now have the following.

**Listing 6.54. Before**

```
class Player { (2)
  // ...
  getX() { return this.x; }
  getY() { return this.y; }
  setX(x: number) { this.x = x; }
  setY(y: number) { this.y = y; }
}
```

**Listing 6.55. After**

```
class Player {
  // ...
  (1)
  moveHorizontal(dx: number) { (2)
    map[this.y][this.x +
       dx].moveHorizontal(this, dx);
  }
  move(dx: number, dy: number) { (2)
    moveToTile(this.x + dx, this.y + dy);
  }
  pushHorizontal(tile: Tile, dx: number) {
       (2)
    if (map[this.y][this.x + dx +
       dx].isAir()
       && !map[this.y + 1][this.x +
       dx].isAir()) {
       map[this.y][this.x + dx + dx] = tile;
       moveToTile(this.x + dx, this.y);
    }
  }
  moveToTile(newx: number, newy: number) {
       (2)
    map[this.y][this.x] = new Air();
    map[newy][newx] = new PlayerTile();
    this.x = newx;
    this.y = newy;
  }
}
```

(1) Getters and setters deleted

(2) New methods pushed into `Player`

Since `moveToTile` was pushed entirely into `Player`, we ***Inline method*** [P4.1.7] on the original `moveToTile`, thereby removing it from the global scope. The new method `Player.moveToTile` is now only called from inside the `Player` class, so we can make it **`private`**. Doing so makes the growing interface for `Player` slightly cleaner.

The process of moving variables and methods into a class is called ***Encapsulate data*** [P6.2.3].

### 6.2.3 *Refactoring:* Encapsulate data

#### DESCRIPTION

As mentioned earlier, we encapsulate variables and methods to limit where they can be accessed from and to make structure explicit. Encapsulating methods helps simplify their names and makes cohesion clearer. This leads to nicer classes – and it often also leads to more and smaller classes, which is beneficial as well. In my experience, people are much too reserved about making classes.

The most significant benefit, however, comes from encapsulating variables. As discussed in chapter 2, we often assume certain properties about our data. These properties become harder to maintain if the data can be accessed from more places. Limiting the scope means only methods inside the class can modify data, and therefore only those methods can affect properties. If we need to verify an invariant, we need only check the code inside the class.

Note that in some situations, we have only methods with a common affix, without variables. It can still make sense to use this refactoring in that situation, but we then need to push the methods into the class before we perform the inner steps.

#### PROCESS

1. Create a class.
2. Move the variables into the new class, replacing **`let`** with **`private`**. Simplify the variables' names; also make getters and setters for the variables.
3. Because the variables are no longer in the global scope, the compiler helps us find all the references by giving errors. Fix these errors in the following five steps:
   a. Pick a good variable name for an instance of the new class.
   b. Replace access with getters or setters on the pretend variable.
   c. If we have errors in two or more different methods, add a parameter with the variable name from earlier as the first parameter, and put the same variable as the first argument at call sites.
   d. Repeat until only one method errors.
   e. If we encapsulated variables, instantiate the new class at the point where the variables were declared. Otherwise, put the instantiation in the method that errors.

#### EXAMPLE

This is a constructed example; it simply increments a variable 20 times, printing the variable's value at every step. Even these few lines are enough to show the potential danger of refactorings similar to this one.

**Listing 6.56. Initial**

```
let counter = 0;
function incrementCounter() {
  counter++;
}
```

```
function main() {
  for (let i = 0; i < 20; i++) {
    incrementCounter();
    console.log(counter);
  }
}
```

We follow the process:

1. Create a class.

**Listing 6.57. New class**

```
class Counter { }
```

2. Move the variables into the new class, replacing **`let`** with **`private`**. Simplify the variables' names; also make getters and setters for the variables.

**Listing 6.58. Before**

```
let counter = 0;
class Counter { }
```

**Listing 6.59. After (1/4)**

```
class Counter {
  private counter = 0; (1)
  getCounter() { return this.counter; } (2)
  setCounter(c: number) { this.counter = c; } (3)
}
```

(1) Encapsulated variable
(2) New getter
(3) New setter

3. Because `counter` is no longer in the global scope, the compiler helps us find all the references by giving errors. Fix these errors in the following five steps:
   a. Pick a good variable name for an instance of the new class: `counter`.
   b. Replace access with getters or setters on the pretend variable.

**Listing 6.60. Before**

```
function incrementCounter()
      {
  counter++;
}
function main() {
  for (let i = 0; i < 20;
      i++) {
    incrementCounter();
    console.log(counter);
  }
}
```

**Listing 6.61. After (2/4)**

```
function incrementCounter() {
  counter.setCounter(counter.getCounter() +
      1); (1)
}
function main() {
  for (let i = 0; i < 20; i++) {
    incrementCounter();
    console.log(counter.getCounter()); (2)
  }
}
```

(1) Assigning replaced with setter
(2) Accessing replaced with getter

   c. If we have errors in two or more different methods, add a parameter with the variable name from earlier as the first parameter, and put the same variable as the first argument at call sites.

**Listing 6.62. Before**

```
function incrementCounter() {

      counter.setCounter(counter.get
      Counter() + 1);
}
function main() {
  for (let i = 0; i < 20; i++) {
    incrementCounter();

      console.log(counter.getCounter
      ());
  }
}
```

**Listing 6.63. After (3/4)**

```
function incrementCounter(counter:
      Counter) { (1)

      counter.setCounter(counter.get
      Counter() + 1);
}
function main() {
  for (let i = 0; i < 20; i++) {
    incrementCounter(counter); (2)

      console.log(counter.getCounter
      ());
  }
}
```

(1) Parameter added

(2) Artificial variable passed as an argument

d. Repeat until only one method errors. In this case, we have only one error at this point.
e. At this point, we can inadvertently make a mistake by initializing the class inside the loop. It is not always easy to know whether the code is somehow run inside a loop. Notice how the following code would not work properly, although it would compile.

**Listing 6.64. Incorrect**

```
function main() {
  for (let i = 0; i < 20; i++) {
    let counter = new Counter(); (1)
    incrementCounter(counter);
    console.log(counter.getCounter());
  }
}
```

(1) Incorrect instantiation location

To make sure we do not make this mistake, we make the distinction of whether we encapsulated variables. In this case, we did, so we instantiate the new class at the point where the variable was.

**Listing 6.65. Before**

```
class Counter { ... }
```

**Listing 6.66. After (4/4)**

```
class Counter { ... }
let counter = new Counter(); (1)
```

(1) Instantiating a variable at the place where the old variable was

After this, we can easily push in `incrementCounter` with the same suffix. The resulting code in this example also breaks one of our rules: can you spot which one and how to fix it?

#### FURTHER READING

This refactoring is very closely related to one called ***Encapsulate field*** that makes a public field private and introduces a getter and setter for it. The difference is that our version also replaces the public access to the field with parameters. This, in turn, allows this pattern to also encapsulate methods without a field.

Converting to parameters has the added benefit that we can more easily move the instantiation around if we see fit. Because of the parameters, we are forced to instantiate the class before we use it, thereby avoiding a possible null reference error that might have occurred when it was globally accessed.

## 6.3 Encapsulating complex data

We have another clear group in the methods and variables:

- `map`
- `transformMap`
- `updateMap`
- `drawMap`

These are asking to be in a `map` class, so we use ***Encapsulate data*** [P6.2.3]:

1. Create a `Map` class.

**Listing 6.67. New class**

```
class Map { }
```

2. Move the variable `map` into `Map`, and replace **`let`** with **`private`**. In this case, we cannot simplify the name. We also make a getter and setter for `map`.

**Listing 6.68. Before**

```
let map: Tile[][];
```

**Listing 6.69. After (1/4)**

```
class Map {
  private map: Tile[][]; ❶
  getMap() { return this.map; } ❷
  setMap(map: Tile[][]) { this.map = map; } ❷
}
```

❶ Move in the variable, changing `let` to `private`

❷ Add a getter and setter for `map`

3. Because `map` is no longer in the global scope, the compiler helps us find all the references by giving errors. We fix these errors in the following five steps:
   a. Pick a good variable name for an instance of the `Map` class: `map`.
   b. Replace access with getters or setters on the pretend variable.

**Listing 6.70. Before**

```
function remove(shouldRemove:
      RemoveStrategy) {
  for (let y = 0; y <
      map.length; y++) {
    for (let x = 0; x <
      map[y].length; x++) {
      if
      (shouldRemove.check(map[y]
      [x])) {
        map[y][x] = new Air();
      }
    }
  }
}
```

**Listing 6.71. After (2/4)**

```
function remove(shouldRemove:
      RemoveStrategy) {
  for (let y = 0; y <
      map.getMap().length; y++) { (1)
    for (let x = 0; x <
      map.getMap()[y].length; x++) { (1)
      if
      (shouldRemove.check(map.getMap()[y]
      [x])) { (1)
        map.getMap()[y][x] = new Air();
      (1)
      }
    }
  }
}
```

(1) Access map through `getMap`

c. If we have errors in or more two different methods, add a parameter with the variable name from earlier as the first parameter, and put the same variable as the first argument at call sites.

**Listing 6.72. Before**

```
interface Tile {
  // ...
  moveHorizontal(player: Player,
      dx: number): void;
  moveVertical(player: Player, dy:
      number): void;
  update(x: number, y: number):
      void;
}
/// ...
```

**Listing 6.73. After (3/4)**

```
interface Tile {
  // ...
  moveHorizontal(map: Map, player:
      Player, dx: number): void; (1)
  moveVertical(map: Map, player:
      Player, dy: number): void; (1)
  update(map: Map, x: number, y:
      number): void; (1)
}
/// ... (2)
```

(1) `map` added as an argument

(2) `map` is added in lots of places.

d. Repeat until only one method errors.
e. We encapsulated a variable, so we put **`let`** `map = new Map();` at the point where `map` used to be.

**Listing 6.74. After**

```
let map = new Map();
```

Resulting in the transformation:

**Listing 6.75. Before**

```
interface Tile {
  // ...
  moveHorizontal(player: Player, dx:
      number): void;
  moveVertical(player: Player, dy:
      number): void;
  update(x: number, y: number): void;
}
/// ...
function remove(shouldRemove:
      RemoveStrategy) {
  for (let y = 0; y < map.length; y++)
      {
    for (let x = 0; x < map[y].length;
      x++) {
      if
      (shouldRemove.check(map[y][x]))
      {
        map[y][x] = new Air();
      }
    }
  }
}
/// ...
let map: Tile[][];
```

**Listing 6.76. After (4/4)**

```
interface Tile {
  // ...
  moveHorizontal(map: Map, player: Player,
      dx: number): void; (1)
  moveVertical(map: Map, player: Player,
      dy: number): void; (1)
  update(map: Map, x: number, y: number):
      void; (1)
}
/// ... (1)
function remove(map: Map, shouldRemove:
      RemoveStrategy) {
  for (let y = 0; y < map.getMap().length;
      y++) { (2)
    for (let x = 0; x <
      map.getMap()[y].length; x++) { (2)
      if
      (shouldRemove.check(map.getMap()[y][
      x])) { (2)
        map.getMap()[y][x] = new Air(); (2)
      }
    }
  }
}
/// ...
class Map {
  private map: Tile[][]; (3)
  getMap() { return this.map; } (3)
  setMap(map: Tile[][]) { this.map = map;
      } (3)
}
```

(1) `map` added as an argument

(2) Accesses map through `getMap`

(3) New class with a getter and setter for `map`

Handling the methods mentioned earlier is now easy: ***Push code into classes*** [P4.1.5] simplifies their names in the process, and we use ***Inline method*** [P4.1.7], as we have done so many times before.

**Listing 6.77. Before**

```
function transformMap(map: Map) {
  map.setMap(new Array(rawMap.length));
  for (let y = 0; y < rawMap.length; y++) {
    map.getMap()[y] = new Array(rawMap[y].length);
    for (let x = 0; x < rawMap[y].length; x++) {
      map.getMap()[y][x] = transformTile(rawMap[y][x]);
    }
  }
}
function updateMap(map: Map) {
  for (let y = map.getMap().length - 1; y >= 0; y--) {
    for (let x = 0; x < map.getMap()[y].length; x++) {
      map.getMap()[y][x].update(map, x, y);
    }
  }
}
function drawMap(map: Map, g: CanvasRenderingContext2D)
      {
  for (let y = 0; y < map.getMap().length; y++) {
    for (let x = 0; x < map.getMap()[y].length; x++) {
      map.getMap()[y][x].draw(g, x, y);
    }
  }
}
```

**Listing 6.78. After**

```
class Map {
  // ...
  transform() {
    this.map = new Array(rawMap.length);
    for (let y = 0; y < rawMap.length; y++) {
      this.map[y] = new Array(rawMap[y].length);
      for (let x = 0; x < rawMap[y].length; x++) {
        this.map[y][x] =
       transformTile(rawMap[y][x]);
      }
    }
  }
  update() {
    for (let y = this.map.length - 1; y >= 0; y--) {
      for (let x = 0; x < this.map[y].length; x++) {
        this.map[y][x].update(this, x, y);
      }
    }
  }
  draw(g: CanvasRenderingContext2D) {
    for (let y = 0; y < this.map.length; y++) {
      for (let x = 0; x < this.map[y].length; x++) {
        this.map[y][x].draw(g, x, y);
      }
    }
  }
}
```

As we did with `Player`, we have a getter and a setter, so we again ***Eliminate getter or setter*** [P6.1.3]. Luckily the setter is unused, so it is trivial to delete. The getter requires some pushing.

**Listing 6.79. Before**

```
class Falling {
  // ...
  drop(map: Map, tile: Tile, x: number, y: number) {
    map.getMap()[y + 1][x] = tile;
    map.getMap()[y][x] = new Air();
  }
}
class FallStrategy {
  // ...
  update(map: Map, tile: Tile, x: number, y: number) {
    this.falling = map.getMap()[y + 1][x].isAir() ? new
      Falling() : new Resting();
    this.drop(map, tile, x, y);
  }
}
/// ...
class Player {
  // ...
  moveHorizontal(map: Map, dx: number) {
    map.getMap()[this.y][this.x + dx].moveHorizontal(map,
      this, dx);
  }
  moveVertical(map: Map, dy: number) {
    map.getMap()[this.y + dy][this.x].moveVertical(map,
      this, dy);
  }
  pushHorizontal(map: Map, tile: Tile, dx: number) {
    if (map.isAir(this.x + dx + dx, this.y)
      && !map.isAir(this.x + dx, this.y + 1)) {
      map.getMap()[this.y][this.x + dx + dx] = tile;
      this.moveToTile(map, this.x + dx, this.y);
    }
  }
  private moveToTile(map: Map, newx: number, newy:
      number) {
    map.getMap()[this.y][this.x] = new Air();
    map.getMap()[newy][newx] = new PlayerTile();
    this.x = newx;
    this.y = newy;
  }
}
/// ...
function remove(map: Map, shouldRemove: RemoveStrategy) {
  for (let y = 0; y < map.getMap().length; y++) {
    for (let x = 0; x < map.getMap()[y].length; x++) {
      if (shouldRemove.check(map.getMap()[y][x])) {
        map.getMap()[y][x] = new Air();
      }
    }
  }
}
class Map {
  // ...
  getMap() { return this.map; }
}
```

**Listing 6.80. After**

```
class Falling {
  // ...
  drop(map: Map, tile: Tile, x: number, y: number)
      {
    map.drop(tile, x, y); (1)
  }
}
class FallStrategy {
  // ...
  update(map: Map, tile: Tile, x: number, y:
      number) {
    this.falling = map.getBlockOnTopState(x, y +
      1); (1)
    this.drop(map, tile, x, y);
  }
}
/// ...
class Player {
  // ...
  moveHorizontal(map: Map, dx: number) {
    map.moveHorizontal(this, this.x, this.y, dx);
      (1)
  }
  moveVertical(map: Map, dy: number) {
    map.moveVertical(this, this.x, this.y, dy); (1)
  }
  pushHorizontal(map: Map, tile: Tile, dx: number)
      {
    if (map.isAir(this.x + dx + dx, this.y) (1)
      && !map.isAir(this.x + dx, this.y + 1)) { (1)
      map.setTile(this.x + dx + dx, this.y, tile);
      (1)
      this.moveToTile(map, this.x + dx, this.y);
    }
  }
  private moveToTile(map: Map, newx: number, newy:
      number) {
    map.movePlayer(this.x, this.y, newx, newy); (1)
    this.x = newx;
    this.y = newy;
  }
}
/// ...
class Map {
  // ...
  (2)
  isAir(x: number, y: number) {
    return this.map[y][x].isAir();
  }
  drop(tile: Tile, x: number, y: number) {
    this.map[y + 1][x] = tile;
    this.map[y][x] = new Air();
  }
  getBlockOnTopState(x: number, y: number) {
    return this.map[y][x].getBlockOnTopState();
  }
```

```
  setTile(x: number, y: number, tile: Tile) {
    this.map[y][x] = tile;
  }
  movePlayer(x: number, y: number, newx: number,
      newy: number) {
    this.map[y][x] = new Air();
    this.map[newy][newx] = new PlayerTile();
  }
  moveHorizontal(player: Player, x: number, y:
      number, dx: number) {
    this.map[y][x + dx].moveHorizontal(this,
      player, dx);
  }
  moveVertical(player: Player, x: number, y:
      number, dy: number) {
    this.map[y + dy][x].moveVertical(this, player,
      dy);
  }
  remove(shouldRemove: RemoveStrategy) { (1)
    for (let y = 0; y < this.map.length; y++) {
      for (let x = 0; x < this.map[y].length; x++)
      {
        if (shouldRemove.check(this.map[y][x])) {
          this.map[y][x] = new Air();
        }
      }
    }
  }
}
```

(1) Code is pushed into `Map`.

(2) `getMap` is removed.

The original `remove` is now a single line, so we use ***Inline method*** [P4.1.7].

Usually we are not fans of introducing such a strong method like `setTile` into our public interface. It very nearly gives complete control to the private field `map`. However, we should not be afraid to add code; we soldier on.

We notice that all the lines but one in `Player.pushHorizontal` use `map`, so we decide to push the code into `map`.

**Listing 6.81. Before**

```
class Player {
  // ...
  pushHorizontal(map: Map, tile: Tile, dx:
      number) {
    if (map.isAir(this.x + dx + dx, this.y)
      && !map.isAir(this.x + dx, this.y + 1)) {
      map.setTile(this.x + dx + dx, this.y,
      tile);
      this.moveToTile(map, this.x + dx, this.y);
    }
  }
  private moveToTile(map: Map, newx: number,
      newy: number) {
    map.movePlayer(this.x, this.y, newx, newy);
    this.x = newx;
    this.y = newy;
  }
}
```

**Listing 6.82. After**

```
class Player {
  // ...
  pushHorizontal(map: Map, tile: Tile, dx: number) {
    map.pushHorizontal(this, tile, this.x, this.y, dx); (1)
  }
  moveToTile(map: Map, newx: number, newy: number) {    (2)
    map.movePlayer(this.x, this.y, newx, newy);
    this.x = newx;
    this.y = newy;
  }
}
class Map {
  // ...
  pushHorizontal(player: Player, tile: Tile, x: number, y:
      number, dx: number) {
    if (this.map[y][x + dx + dx].isAir()
      && !this.map[y + 1][x + dx].isAir()) {
      this.map[y][x + dx + dx] = tile;
      player.moveToTile(this, x + dx, y);
    }
  }
}
```

(1) Code pushed into `Map`

(2) Method is made public

This `setTile` is only used inside `Map`. We can make it private or — even better — remove it, since we love deleting code.

## 6.4 Eliminating a sequence invariant

We notice that the map is initialized with a call to `map.transform`. But in an object-oriented setting, we have a different mechanism for initialization: the constructor.

In this case, we are lucky because we can replace `transform` with **`constructor`** and remove the call to `transform`.

**Listing 6.83. Before**

```
class Map {
  // ...
  transform() {
    // ...
  }
}
/// ...
window.onload = () => {
  map.transform();
  gameLoop(map);
}
```

**Listing 6.84. After**

```
class Map {
  // ...
  constructor() { (1)
    // ...
  }
}
/// ...
window.onload = () => {
  (2)
  gameLoop(map);
}
```

(1) `transform` changed to **`constructor`**

(2) Call to `transform` removed

Doing this has the significant effect of removing the invariant that we have to

call `map.transform` before the other methods. When something needs to be called before something else, we call it a *sequence invariant*. It is impossible not to call the constructor first, so the invariant is eliminated. This technique can always be used to make sure things happen in a specific sequence. We call this refactoring ***Enforce sequence*** [P6.4.1].

### 6.4.1 *Refactoring:* Enforce sequence

#### DESCRIPTION

I think the coolest type of refactoring is when we can "teach" the compiler something about how we want our program to run, so it can help make sure that happens. This is one of those situations.

Object-oriented languages have a built-in property that constructors are always called before methods on objects. We can take advantage of this property to make sure things happen in a specific order. It is even fairly straightforward to do, although it means introducing one class per step that we want to enforce. But after performing this transformation, the sequence is no longer an invariant, because it is enforced! We don't need to remember to call one method before the other, because it is impossible not to do so. Amazing!

By using the constructor to ensure that some code is run, the instance of the class becomes proof that the code was run. We cannot get an instance without running the constructor successfully.

This example shows how to use this technique to make sure a string is capitalized before it is printed.

**Listing 6.85. Before**

```
function print(str: string) {
  // string should be capitalized
  console.log(str);
}
```

**Listing 6.86. After**

```
class CapitalizedString {
  private value: string;
  constructor(str: string) {
    this.value = capitalize(str);
  }
  print() {
    (1)
    console.log(this.value);
  }
}
```

(1) The invariant disappeared.

The ***Enforce sequence*** transformation has two variants: internal and external. The previous example demonstrates the internal version: the target function is moved inside the new class. Here is a side-by-side comparison of the two variants, which mostly offer the same advantages.

**Listing 6.87. Internal**

```
class CapitalizedString {
  private value: string; (1)
  constructor(str: string) {
    this.value = capitalize(str);
  }
  print() { (2)
    console.log(this.value);
  }
}
```

**Listing 6.88. External**

```
class CapitalizedString {
  public readonly value: string; (1)
  constructor(str: string) {
    this.value = capitalize(str);
  }
}
function print(str: CapitalizedString) { (2)
  console.log(str.value);
}
```

(1) Private vs. public

(2) Method vs. function with a specific parameter type

This refactoring pattern focuses on the internal version because it leads to stronger encapsulation by not having a getter or a public field.

### PROCESS

1. Use ***Encapsulate data*** [P6.2.3] on the method that should run last.
2. Make the constructor call the first method.
3. If arguments of the two methods are connected, make these arguments into fields, and remove them from the method.

### EXAMPLE

Let's look at an example similar to the earlier one about a bank. We want to make sure money is always first subtracted from the sender before it's added to the receiver. The sequence is thus a `deposit` with a negative amount followed by a `deposit` with a positive amount.

**Listing 6.89. Initial**

```
function deposit(to: string, amount: number) {
  let accountId = database.find(to);
  database.updateOne(accountId, { $inc: { balance: amount } });
}
```

1. Use ***Encapsulate data*** [P6.2.3] on the method that should run last.

**Listing 6.90. Before**

```
function deposit(to: string, amount:
      number) {
  let accountId = database.find(to);
  database.updateOne(accountId, {
      $inc: { balance: amount } });
}
```

**Listing 6.91. After (1/2)**

```
class Transfer { (1)
  deposit(to: string, amount: number)
      {
    let accountId = database.find(to);
    database.updateOne(accountId, {
      $inc: { balance: amount } });
  }
}
```

(1) New class

2. Make the constructor call the first method.

**Listing 6.92. Before**

```
class Transfer {
  deposit(to: string, amount: number)
      {
    let accountId =
      database.find(to);
    database.updateOne(accountId, {
      $inc: { balance: amount } });
  }
}
```

**Listing 6.93. After (2/2)**

```
class Transfer {
  constructor(from: string, amount:
      number) { (1)
    this.deposit(from, -amount); (1)
  } (1)
  deposit(to: string, amount: number)
      {
    let accountId =
      database.find(to);
    database.updateOne(accountId, {
      $inc: { balance: amount } });
  }
}
```

(1) New constructor calling the first method

We have now guaranteed that `deposit` is called with a negative `amount` from the sender, but we can go further. We can connect the two `amounts` by making this argument a field and removing `amount` from the method. Because we need it to be negated in one case, we introduce a helper method. The result looks like the following.

**Listing 6.94. After**

```
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);
  }
}
```

We have made sure that we cannot create money, but money can disappear if we forget to call `deposit` with a receiver. Therefore, we might want to wrap this class in another class to ensure that a positive transfer also occurs.

#### FURTHER READING

I am not familiar with any formal description of a pattern like this. There are undoubtedly people familiar with this way of using objects as proof that something has happened, but I have not come across such a discussion.

## 6.5 Dealing with enums

One last method that feels distinct is `transformTile`, because of the `Tile` suffix. We already have a class (or, more specifically, an enum) with the same suffix: `RawTile`. The name `transformTile` suggests that this method should be moved to the `RawTile` *enum*. However, this is not possible in many languages, including

TypeScript: **`enum`**s cannot have methods.

### 6.5.1 Enumeration through private constructors

If our language does not support methods on enums, there is a technique we can use to get around that by using a private constructor. Every object must be created by invoking a constructor. If we make the constructor **`private`**, objects can only be created inside our class. Specifically, we can control how many instances exist. If we put these instances in public constants, we can use them as enums:

**Listing 6.95. Enum**

```
enum TShirtSize {
  SMALL,
  MEDIUM,
  LARGE,

}
function sizeToString(s: TShirtSize) {
  if (s === TShirtSize.SMALL) return
       "S";
  else if (s === TShirtSize.MEDIUM)
       return "M";
  else if (s === TShirtSize.LARGE)
       return "L";
}
```

**Listing 6.96. Private constructor**

```
class TShirtSize {
  static readonly SMALL = new
       TShirtSize();
  static readonly MEDIUM = new
       TShirtSize();
  static readonly LARGE = new
       TShirtSize();
  private constructor() {}
}
function sizeToString(s: TShirtSize) {
  if (s === TShirtSize.SMALL) return
       "S";
  else if (s === TShirtSize.MEDIUM)
       return "M";
  else if (s === TShirtSize.LARGE)
       return "L";
}
```

The only exception is that we cannot use **`switch`** with this construction, but we have a rule preventing us from doing that anyway. Note that some weird behavior happens if we serialize and deserialize our data, but that is out of the scope of this book.

Now `TShirtSize` is a class (which is awesome), and we can push code into it. Unfortunately, we cannot simplify away the **`if`**s in this setup, because unlike last time, we do not have a class for each value: we have only one class. To gain the full benefit, we need to remedy this situation: we need to ***Replace type code with classes*** [P4.1.3].

**Listing 6.97. Enum**

```
interface SizeValue { }
class SmallValue implements SizeValue { ... }
class MediumValue implements SizeValue { ... }
class LargeValue implements SizeValue { ... }
```

Again, we could simplify these names with namespaces or packages. We can skip the `is` methods this time because we never create new instances on the fly, so `===` is enough. We then use these new classes as an argument for each value in the private-constructor class. We also store it as a field.

**Listing 6.98. Before**

```
class TShirtSize {
  static readonly SMALL = new
      TShirtSize();
  static readonly MEDIUM = new
      TShirtSize();
  static readonly LARGE = new
      TShirtSize();
  private constructor() {}
}
```

**Listing 6.99. After**

```
class TShirtSize {
  static readonly SMALL = new TShirtSize(new
      SmallValue()); (1)
  static readonly MEDIUM = new TShirtSize(new
      MediumValue()); (1)
  static readonly LARGE = new TShirtSize(new
      LargeValue()); (1)
  private constructor(private value: SizeValue)
      {} (2)
}
```

(1) Passing new classes as arguments

(2) Parameter and field for the values

Now, whenever we push something into `TShirtSize`, we can push it further into all the classes and resolve `=== TShirtSize.`, thereby getting rid of the **`if`**s. This could have been a pattern, but I have chosen not to make it one for two reasons. First, this process does not apply equally to all programming languages – in particular, Java. Second, we already have a pattern for eliminating enums, which should take preference.

In the game, one enum remains: `RawTile`. We have already performed ***Replace type code with classes*** [P4.1.3] on it, but we could not eliminate this enum since we use the indices in places. However, we can use the previous transformation to eliminate it anyway.

We introduce a new `RawTile2` class with a private constructor with a field for each value of the enum. We also create a new `RawTileValue` interface and classes for each of the values of the enum, which we pass as arguments for the fields in `RawTile2`.

**Listing 6.100. New class**

```
interface RawTileValue { }
class AirValue implements RawTileValue { }
// ...
class RawTile2 {
  static readonly AIR = new RawTile2(new AirValue());
  // ...
  private constructor(private value: RawTileValue) { }
}
```

We are one step closer to eliminating the enum. Now we need to switch to using the classes instead of the enums.

### 6.5.2 Remapping numbers to classes

In some languages, enums cannot have methods because they are handled like named integers. In our game, we store our `rawMap` as integers and then can interpret the integers as enums. To replace the enums, then we need a way to convert the numbers to our new `RawTile2` instances. The easiest way to do this is to make an array with all the values in the same order as in the enum.

**Listing 6.101. Before**

```
enum RawTile {
  AIR,
  FLUX,
  UNBREAKABLE,
  PLAYER,
  STONE, FALLING_STONE,
  BOX, FALLING_BOX,
  KEY1, LOCK1,
  KEY2, LOCK2
}
```

**Listing 6.102. After**

```
const RAW_TILES = [
  RawTile2.AIR,
  RawTile2.FLUX,
  RawTile2.UNBREAKABLE,
  RawTile2.PLAYER,
  RawTile2.STONE, RawTile2.FALLING_STONE,
  RawTile2.BOX, RawTile2.FALLING_BOX,
  RawTile2.KEY1, RawTile2.LOCK1,
  RawTile2.KEY2, RawTile2.LOCK2
];
```

With this, we can easily map numbers to the correct instance. With `RawTile` gone, we change the remaining references of `RawTile` to `RawTile2`-- or, if that is impossible, to `number`.

**Listing 6.103. Before**

```
let rawMap: RawTile[][] = [
  // ...
];
class Map {
  private map: Tile[][];
  constructor() {
    this.map = new
      Array(rawMap.length);
    for (let y = 0; y <
      rawMap.length; y++) {
      this.map[y] = new
      Array(rawMap[y].length);
      for (let x = 0; x <
      rawMap[y].length; x++) {
        this.map[y][x] =
      transformTile(rawMap[y][x]);
      }
    }
  }
  // ...
}
function transformTile(tile:
      RawTile) {
  // ...
}
```

**Listing 6.104. After**

```
let rawMap: number[][] = [ (1)
  // ...
];
class Map {
  private map: Tile[][];
  constructor() {
    this.map = new Array(rawMap.length);
    for (let y = 0; y < rawMap.length; y++) {
      this.map[y] = new
      Array(rawMap[y].length);
      for (let x = 0; x < rawMap[y].length;
      x++) {
        this.map[y][x] =
      transformTile(RAW_TILES[rawMap[y][x]]);
      (2)
      }
    }
  }
  // ...
}
/// ...
function transformTile(tile: RawTile2) { (3)
  // ...
}
```

(1) Impossible to put `RawTile2`

(2) Maps the number to the class

(3) Parameter changed to a class

Now we get an error in `transformTile`. The **`switch`** that remains from earlier is an issue because as mentioned, the private constructor method does not work with **`switch`**. All this work was to eliminate the enum and with it this **`switch`**. We therefore ***Push code into classes*** [P4.1.5], through `RawTile2` and into all the classes.

**Listing 6.105. Before**

```
interface RawTileValue { }
class AirValue implements
      RawTileValue { }
class StoneValue implements
      RawTileValue { }
class Key1Value implements
      RawTileValue { }
/// ...
class RawTile2 {
  // ...
}
/// ...
function assertExhausted(x: never):
      never {
  throw new Error("Unexpected object:
      " + x);
}
function transformTile(tile:
      RawTile2) {
  switch (tile) {
    case RawTile.AIR: return new
      Air();
    case RawTile.STONE: return new
      Stone(new Resting());
    case RawTile.KEY1: return new
      Key(YELLOW_KEY);
    // ...
    default: assertExhausted(tile);
  }
}
```

**Listing 6.106. After**

```
interface RawTileValue {
  transform(): Tile;
}
class AirValue implements RawTileValue {
  transform() { return new Air(); }
}
class StoneValue implements RawTileValue {
  transform() { return new Stone(new
      Resting()); }
}
class Key1Value implements RawTileValue {
  transform() { return new Key(YELLOW_KEY);
      }
}
/// ...
class RawTile2 {
  // ...
  transform() {
    return this.value.transform(); (1)
  }
}
(2)
function transformTile(tile: RawTile2) {
  return tile.transform();
}
```

(1) The code is pushed right through into the values.

(2) The magical `assertExhausted` is no longer needed.

At last, the switch has disappeared. `transformTile` is a single line, so we ***Inline method*** [P4.1.7]. Finally, we rename `RawTile2` to its permanent name: `RawTile`.

## 6.6 Summary

- To help enforce encapsulation, avoid exposing data. The rule ***Do not use getters and setters*** [R6.1.1] states that we should not expose private fields indirectly through getters and setters either. We can use the refactoring pattern ***Eliminate getter or setter*** [P6.1.3] to get rid of getters and setters.
- The rule ***Never have common affixes*** [R6.2.1] states that if we Have methods and variables with a common prefix or suffix, they should be in a class together. We can use the refactoring pattern ***Encapsulate data*** [P6.2.3] to achieve this.
- By using classes, it is possible to make the compiler enforce a sequence invariant, thereby eliminating it with the refactoring ***Enforce sequence*** [P6.4.1].
- Another method for dealing with enums is to use a class with a private constructor. Doing so can further eliminate enums and switches.

This concludes part 1 of the book. We can continue to encapsulate things, like `inputs` and `handleInputs`; we can even encapsulate `player` and `map` in a `Game` class, but I'll leave that to you.

We can also extract constants, improve variable and method naming, and introduce namespaces, or go all out on type codes and convert some or all booleans to enums and then ***Replace type code with classes*** [P4.1.3] — and thus the snowball has started rolling. The point is, this is not the end of the refactoring. Rather, it is a strong start! In part 2 of the book, we discuss some of the general principles that enable us to do great refactoring.

I claim that everything we've done with the example game in part 1 has already resulted in a much better architecture for three primary reasons:

1. It is now much quicker and safer to extend the game with new `Tile` types.
2. It is much easier to reason about the code because related variables and functionality are grouped in classes and methods with helpful names.
3. We now have control over the scope of our data, with much finer granularity. Therefore it is harder to program something that breaks non-local invariants – which, as discussed in chapter 2, is the cause of most bugs.

In a few places, we have investigated the code a bit to give things good names or decide whether elements should stay together. But these investigations were quick: we never had to spend time figuring out weird quirks in the code, like why one of the **`for`** loops in `update` goes backward, or why we push the inputs on a stack instead of executing the moves directly (we might not even have noticed the stack). Answering questions like these requires much more time to gain an understanding that we didn't require for our refactoring efforts.

# 7 Work With the Compiler

In part 2, we look deeper into how we 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 strengths and weaknesses of compilers
- Using the strengths to eliminate invariants
- Sharing responsibility with the compiler

When we are just learning programming the compiler can feel like an endless source of nagging and nit-picking. It takes things too literally, it has 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 some 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 therefore start by getting to know our compiler. Getting a firm grip of what it can so we can actively use it and build on top of its strengths. And similarly 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 of correctness with it, letting it to 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 very little benefit.

Once we have accepted sharing the responsibility, we need to trust the compiler. We need to make an effort to keep dangerous invariants to a minimum, and we need to listen to its output. That includes its warnings.

The final stage of this journey is accepting that the compiler is better at predicting our programs 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 most powerful tool, so we should use it!

## 7.1 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, certainly the mainstream Java, C#, TypeScript variety.

Our compiler is a program, it is good at certain things, like consistency; contrary to common folk lore compiling again will not yield a different result. Likewise it is bad at certain things, like judgement; compilers follow the common idiom "When in doubt: ask."

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

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 experimentation 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?

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

### 7.1.1 Weakness: Halting problem

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 it will behave, and even then we only observe one path through our program.

***Halting problem***
In general programs are fundamentally unpredictable.

For a quick demonstration of why this is true, consider the program:

**Listing 7.1. Program without runtime error**

```
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 five.

Some programs will definitely fail and are rejected. Some will definitely not and are allowed. The halting problem means that 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 it is safe, this is called a *conservative* analysis.

The conservative analyses prove that there is no possibility for some specific failure in our program. These we can rely on, not the others.

### 7.1.2 Strength: Reachability

One of the conservative analyses checks whether a method **`return`**s 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 this looks like a runtime error, the `never` keyword is forcing the compiler to analyse 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); ❶
}
```

❶ 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 difficult analysis to take advantage of. Especially following the five lines rule, since then it is easy to spot how many and where the **`return`**s are.

### 7.1.3 Weakness: Null checks

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 languages can detect some of these cases, but they can rarely detect all, which means we cannot rely on it 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 7.3. 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`. Rather 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 possibly catch an error.

### 7.1.4 Strength: Definite assignment

Another property that compilers are good at verifying is whether variables have definitely been assigned values before they are used. Note, that does not mean that they have anything useful in them, 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 7.4. 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 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; readonly (or final) fields. A readonly 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 certain values exist. In the example from above we can wrap our array in a class with a readonly 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, but by making this change we prevent anyone from ever causing the John-object to disappear unnoticed, thereby eliminating an invariant.

### 7.1.5 Weakness: Arithmetic errors

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

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

**Listing 7.5. No compiler error**

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

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

### 7.1.6 Weakness: Out of bounds errors

Yet another place where the compiler is in hot water is when we access directly into 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 we have a function to find the index of the first prime in an array, we may use this function to find the first prime like this:

**Listing 7.6. No compiler error**

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

However, if there is no prime in the array such a function would 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 method from definite assignment above, to prove that the element definitely exists.

### 7.1.7 Strength: Access control

Something the compiler is excellent at is access control, which we have used when we have encapsulated data. If we make a member private we can be certain that it does not

escape accidentally. As we have already seen plenty examples of how and why to use this in chapter 6 we will not go into further detail with it here, except for clearing up a common misconception among junior programmers: **`private`** applies to the class *not* the object. This means that we can inspect another objects 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 like this:

**Listing 7.7. Compiler error due to access**

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

❶ Compiler error here

### 7.1.8 Weakness: Termination checking

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 provide no assistance for this kind of error.

In this example we want to detect whether we are inside a string or not. 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 this.

**Listing 7.8. No compiler error**

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

The way these are being reduced is by transitioning away from **`while`**, to **`for`** then `foreach`, and lately to more high level constructions such as `forEach` in TypeScript, stream operation in Java, and Linq in C#.

### 7.1.9 Strength: Type checking

The final strength of compiler 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, which we have used whenever we have renamed something to get errors in part one. It was also the type checker that enabled ***Enforce sequence*** [P6.4.1].

In this example we have encoded a list data structure which cannot be empty, because

it can only be made up of one element or an element or the rest of the list.

**Listing 7.9. 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([]); ❶
```

❶ 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 we consider in this book limits its type strength to be equivalent with 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. Although this requires some mastery of type theory. Several languages have even stronger type systems, the most interesting of which in generally increasing order of strength:

- Borrowing types of Rust
- Polymorphic type inference of OCaml and F#
- Type classes of Haskell
- Union and intersection types of 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 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 peaked the readers' interest enough for them to seek it out on their own.

### 7.1.10 Weakness: Deadlocks and race conditions

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

TypeScript does not support multiple threads, so I cannot make examples of these errors in TypeScript. However I can give a pseudo code demonstration of them.

Race condition is the first problem with threads we run into. 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:

**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 (1)
5 (1)
7 (2)
8
...
```

(1) Both repeating numbers

(2) And skipping numbers

To solve this issue we introduce locks. Let's give each thread a lock, and check that the other threads 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(a, b));
let b = new Thread(new Counter(b, a));
a.start();
b.start();
```

**Listing 7.13. Example output**

```
1
2
3
4
(1)
```

(1) Nothing happens

The problem we have just stumbled upon is called a deadlock; where both thread are locked waiting for the other to unlock before continuing. A common metaphor for this is two people meeting at a door and both insist that the other is allowed to walk 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:

**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", a, b));
let b = new Thread(new Printer("B", b, a));
a.start();
b.start();
```

**Listing 7.15. Example output**

```
A
A
A
A
(1)
```

(1) Continues for ever

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 seizes.

Entire books are written about how to manage these issues. The best advice I can give to help alleviate all of these 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 Use 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 should certainly not fight or cheat the compiler.

Often people draw similarities between software development and construction. But as Martin Fowler has noted on his blog this is one of the most damaging metaphor's 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 out 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 who's name eludes me

Dan North has noted the similarity that programs are the development teams' 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 Gain safety

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.

#### 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 wanted to change something we simply rename the source method and rely on the compiler to tell us everywhere else we need to do something. Doing it this way we are safe in the knowledge that the compiler does not miss anywhere. This works well, but only when we don't have other errors.

Imagine we want to find everywhere 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 to the name, like `_handled`. Now the compiler errors everywhere we use the enum. And once we have handled a place we can simply append `_handled` to get rid of the error.

**Listing 7.16. Finding enum usages by compiler errors**

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

❶ Compiler errors

Once we are done we can easily remove the `_handled` from everywhere.

#### ENFORCE SEQUENCE

The pattern ***Enforce sequence*** [P6.4.1] is dedicated to teaching the compiler about an invariant of our program thereby making it a property instead. This means the invariant can no longer be accidentally broken somewhen in the future, because the compiler guarantees the property still holds every time we compile.

In chapter 6 we discussed both an internal and external variant of using classes to enforce sequences. These classes both guarantee that a string has at some prior point

has been capitalized.

**Listing 7.17. Internal**

```
class CapitalizedString {
  private value: string; (1)
  constructor(str: string) {
    this.value = capitalize(str);
  }
  print() { (2)
    console.log(this.value);
  }
}
```

**Listing 7.18. External**

```
class CapitalizedString {
  public readonly value: string; (1)
  constructor(str: string) {
    this.value = capitalize(str);
  }
}
function print(str: CapitalizedString) { (2)
  console.log(str.value);
}
```

(1) Private vs. public

(2) Method vs. function with a specific parameter type

### ENCAPSULATION

By using the compilers 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 we could prevent someone from accidentally calling a helper method `depositHelper`, by making it private:

**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);
  }
}
```

### DETECT UNUSED CODE

We have also used the compiler to check whether code was unused with the refactoring pattern ***Try delete then compile*** [P4.5.1]. Deleting a flurry of methods at once the compiler can quickly scan through our entire codebase and let us know which of the methods were in fact used.

We use this to get rid of methods in interfaces. The compiler cannot know whether they are to-be-used or truly 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 it.

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();
```

#### DEFINITE VALUE

Finally, earlier in this chapter we showed a list data structure that could not be empty. We guarantee this by using readonly fields. These are within the compilers definite assignment analysis, and have to 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 readonly fields:

**Listing 7.21. Non-empty list due to readonly 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 Do not fight it

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 offences, each with a section dedicated to it below: not understanding types, being lazy, and not understanding architecture.

#### TYPES

As described above the type checker is the strongest part of the compiler. Therefore tricking it or disabling it is the worst offence. 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.

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 needs to be learned, but needing a cast is a symptom that either we don't understand the types, or someone else didn't.

We need a cast when some type is not what we need it to be. Putting 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 get some 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 situations here, either we get the input from somewhere we have control over, ie. it is our own web service. A more permanent solution would be to somehow reuse the same types on the sending side as the receiving side — there are several libraries to assist with this. If the input comes from a third party, the most safe 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 this is allowed by using `any` (`dyn` in C#). While this may seem useful especially when working sending JSON object back and forth over HTTP, it opens up to a myriad of potential errors, such as referring fields that don't exist, or have different types than we expect, such 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 allows any call on it:

**Listing 7.23. Using any**

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

It is unlikely that this method will not be present at runtime, so it is 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 advice in this book. Here is a fairly common example of how this happens. Imagine we have a method with ten parameters. This is confusing and every time we add or remove one we need to correct it everywhere the

method is called. We decide to instead of taking ten parameter we take one parameter; a Map from strings to values. Then we can easily just 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 the type checking, to the weakness of out of bounds errors. Tired of laundry? Easy solution: just burn all your clothes.

In this example 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 offence is laziness. I don't feel like programmers are to blame suffering from being lazy, since it is what got most people into programming in the first place. We happily spend hours or weeks tirelessly working to automating something we are too lazy to do. Being lazy makes us to better programmers; staying lazy makes us worse programmers.

Another reason for my leniency in this offence is that developers are often under tremendous stress and tight deadlines to deliver. In that state of mind anyone takes the shortcuts they can. The problems is: they are short term fixes.

### Defaults

We have discussed default values quite a bit in part one. Wherever we put 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"); (1)
```

(1) John is now female

### Inheritance

Through the rule ***Only inherit from interfaces*** [R4.3.2] 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 section above. Further though inheritance also adds coupling between its implementing classes.

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

**Listing 7.27. Problem due to inheritance**

```
class Mammal {
  laysEggs() { return false; }
}
class Dolphin extends Mammal { }
/// ...
class Platypus extends Mammal {
  (1)
}
```

(1) Should have overwritten `laysEggs`

### Unchecked exceptions

Exceptions often come in two flavors; those we are forced to handle, and those which 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 only use unchecked exceptions for things that cannot happen, such as when we know some invariant that we cannot express in the language. Having one unchecked exception called `Impossible` seems sufficient. But like with all invariants we risk that one day it is broken, and we 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 case 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 unchecked exception**

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

A better solution is to use a checked exception, and if a local invariant at the call-site guarantees that the exception cannot happen she can easily use the `Impossible` exception mentioned above. This is pseudo code, as Typescript, unfortunately, does not have checked exceptions:

**Listing 7.29. Using unchecked exception**

```
class Impossible extends RuntimeException { }
class EmptyArray extends CheckedException { }
function average(arr: number[]) throws EmptyArray {
  if (arr.length) 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 because they do not understand architecture, specifically micro-architecture. Micro-architecture is architecture that effects this team, but not other teams.

The main way this comes to fruition we have also already discussed in part one, namely breaking encapsulation with getters and setters. This 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 that external code can depend on it. Even worse it can change the stack, by changing the array.

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

```
class Stack<T> {
  private data: T[];
  getArray() { return this.data; }
}
stack.getArray()[0] = newBottomElement; (1)
```

(1) This line changes the stack

Another way this can happen is if we pass a private field as an argument, which has the exact 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 parameter**

```
class Stack<T> {
  private data: T[];
  printLast() { printFirst(this.data); }
}
function printFirst<T>(arr: T[]) {
  arr[0] = newBottomElement; (1)
}
```

(1) This line changes the stack

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

## 7.3 Trust the compiler

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

We can get away with the counter productive 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 it invariants

The malice of global invariants have been discussed at length throughout the book, so these should be under control by now. But what about the 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 us look at a bit larger example where this comes into play. In the example we are creating a data structure to count element. 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.

**Listing 7.32. Counting set**

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

(1) Keeping track of total

We want to add a method for picking a random element out of this data structure. We can 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.

**Listing 7.33. Pick random element (error)**

```
class CountingSet {
  // ...
  randomElement(): string { (1)
    let index = randomInt(this.total);
    for (let key in this.data.keys()) {
      index -= this.data[key];
      if (index <= 0)
        return key;
    }
  }
}
```

(1) Error due to reachability

However this method doesn't compile, since we fail the reachability analysis described above. 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:

**Listing 7.34. Pick 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(); (1)
  }
}
```

(1) Exception to avoid error

However, this only solves the immediate issue of the compiler complaining, we have not added any security that this invariant is not 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 our 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 earth-bound can help you.

"Can't" in this context means that it is infeasible, rather than necessarily 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, because tests tell you when they grow out of sync with the software, documentation offers no such curtesy. Similar for each of the four, the higher on the list, the cheaper it is in the long term.

This should disarm the all too common excuse that we have no time for writing tests, as not doing it is sure to be more time expensive in the long term. If your software has a short lifetime you can permit yourself to select an option lower on the list, like when building a prototype that is to be thrown out after presentation testing it manually is quite sufficient.

### 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, when doctors simply get desensitized to the noises due to them being 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, that states: if something is in pristine condition, people strive 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 of the warnings are unjustified, the danger is that we might miss a crucial one, because it is drowned by the insignificant ones. This is one of the most important 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, it is to help us make fewer mistakes. Therefore, there is only one number of warnings that 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 to the number of warnings we allow in our codebase and then decrease this number bit by bit every month. This is a daunting task, especially since we wont 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 veer their ugly heads again.

## 7.4 Trust 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 indeed 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 own judgement we can be satisfied with the compilers. Instead of straining ourselves wondering whether something will work or not, we can just as the compiler.

If we have taught our compiler the structure of our domain, encoded the invariants, and are used to a 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 it cannot know whether our code solves the problem we expect it to, but it can tell us whether it 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, ie. the programming language. Indeed this quote includes the compiler:

> *If you're the smartest person in the room, you're in the wrong room.*
>
> — Origin unknown

## 7.5 Summary

- Know the strengths and weaknesses common to modern compiler, we can adjust

our code such as to avoid the weaknesses and take advantage of the strengths.

- Learn to use the compiler instead of fighting it, to reach higher levels of safety.
- 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 or not.

# 8 Avoid Comments

***This chapter covers:***

- Understanding the danger of comments
- Identifying comments that add value
- Dealing with different types of comments

Comments are probably one of the most controversial topics in this book. So let's start by clearing up which comments we are talking about. We consider in this chapter comments that are inside methods, and *not* used by external tools, like JavaDoc.

### *Useful documentation*

```
interface Color {
  /**
   * Method for converting a color to a hex string.
   * @returns a 6 digit hex number prefixed with hashtag
   */
  toHex(): string;
}
```

Although controversial to some my opinion align almost perfectly with those expressed by many brilliant programmers. Comments are an art form, unfortunately not many study how to write good comments. Consequently they end up writing only poor comments, which devalues the code. Therefore as a general rule I recommend avoiding them. Rob Pike presented similar arguments back in 1989 in his "Notes on Programming in C":

> *[Comments are] a delicate matter, requiring taste and judgement. I tend to err on the side of eliminating comments, for several reasons. First, if the code is clear, and uses*

> *good type names and variable names, it should explain itself. Second, comments aren't checked by the compiler, so there is no guarantee they're right, especially after the code is modified. A misleading comment can be very confusing. Third, the issue of typography: comments clutter code.*
>
> — Rob Pike

Martin Fowler extends this listing comments as a smell. One of his arguments being that they are often used a deodorant on top of otherwise smelly code. Instead of adding comments, we should clean the code.

Many educators demand the students explain their code through comments, so we learn to write comments right from the start. This is like putting intermediate calculations for a an assignment; good for education, but less useful in the real world. Carrying over this idea to the real world runs into a problem. The issue of incomprehensible code is likely not solved by having the same developer add comments, as expressed in this tweet by Kevlin Henney:

> *A common fallacy is to assume authors of incomprehensible code will somehow by able to express themselves lucidly and clearly in comments.*
>
> — Kevlin Henney

Comments are not checked by the compiler making them easier to write than code, since there are no constraints on them. However, exactly because the compiler does not know about them in systems with a long life span they have a tendency to grow out of date, becoming either irrelevant, or worse: downright misleading.

There are many uses for comments from planning your work, indicating 'hacks', documenting, to removing code. In Robert C. Martin's "Clean Code" he names around 20 types of comments. So many categories can be quite overwhelming to keep track of. Here we split comments into five different categories, each with a specific suggestion on how to approach it.

In most cases we should avoid comments in the code that we deliver. Intermediate comments are great! Therefore comments should be dealt with in the refactoring phase of our workflow. Before delivering any comment always consider if there is a better way to express what it says. We would love to make a rule saying never to use them, but in some cases a comment can save us from making expensive mistakes, in which case they are usually worth their expense. Some properties are difficult or expensive to enforce through code, but can be expressed in a comment in seconds. This is a similar take on comments to Kevlin Henney's approach summarized as:

> *Comment only what the code cannot say.*
>
> — Kevlin Henney

The five categories are ordered from easiest solution to hardest. Let's get into it.

## 8.1 Out-dated comments should be deleted

Here we are generous with our wording, because this category also includes downright wrong or misleading comments. We do so with the justification that likely the comment was well intended when written, but then grew out of sync with the codebase.

In this example notice how the comment and the condition disagrees on whether it is "or" or "and". This can be dangerous.

**Listing 8.1. Out-dated comment**

```
if (element.hasSelection() || element.isMultiSelect()) {
  // Is has a selection and allows multi selection
  // ...
}
```

The easiest type of comment to deal with are those that have gone out of date. This means that the comment is now either irrelevant or incorrect. These comments do not save us any time, but they take time to read, so we should delete them.

The worse effect of these comments though is when they mislead us. Not only do we waste time reading them, but if we design our code relying on something untrue, we may have to do considerable rework. But worst of all they can cause us to introduce bugs in the code.

## 8.2 Out-commented code should be deleted

Sometimes we experiment with removing some code, in which case it can be quick and easy to comment it out and see what happens. This is a good way to experiment. But after our experiment we should delete any out-commented code. Since our code is in version control it is easy to recover even after we delete it.

In this example it is easy to see why the comments ended there: there was a first draft of the code, working, but suboptimal. A developer thought he could improve it, but not confident of his success — understandably because it is not an easy algorithm — and not supported by his abilities in version control, either due to inexperience or because branching is expensive. Therefore, instead of deleting the old algorithm, he simply out-commented it, so he could quickly revert if the new algorithm couldn't be brought to function. To test whether it was working he might have had to merge it with `master`, and when it tested successfully it was already there, and there was no time, nor reason to meddle with something that was working.

**Listing 8.2. Out-commented code**

```
const PHI = (1 + Math.sqrt(5)) / 2;
const PHI_ = (1 - Math.sqrt(5)) / 2;
const C = 1 / Math.sqrt(5);
function fib(n: number) {
  // if(n <= 1) return n;
  // else return fib(n-1) + fib(n-2);
  return C * (Math.pow(PHI, n) - Math.pow(PHI_, n));
```

```
}
```

A way this scenario should have played out is: The developer creates a branch in Git, deletes the old code, and starts working on the new. If he discovers that it cannot work, he checkouts the `master` branch, and deletes the one he created for the experiment. If it works, he merges with `master` and everything is clean. Even with the requirement of merging into `master` to test, we follow this procedure, and if then the code cannot work, we recover the original code from the version history.

## 8.3 Trivial comments should be deleted

Another category is when comment do not add anything. When the code is as easy to read as the comment we say the comment is *trivial*.

**Listing 8.3. Trivial comment**

```
/// Log error
Logger.error(errorMessage, e);
```

In this category we also include comments that we ignore when we scan over the code. If no one ever reads a comment it is just taking up space, and we can get rid of it for free.

## 8.4 Comments that could be a method name

Some comments document the code rather than the functionality. This is easiest explained with an example:

**Listing 8.4. Comment documenting the codes**

```
/// Build request url
if (queryString)
  fullUrl += "?" + queryString;
```

In these cases we can simply extract the block into a method with the same name as the comment. As seen below, after this operation the comment is trivial, and we deal with it accordingly; we delete it. We saw this solution being utilized twice way back in chapter 3.

**Listing 8.5. Before**

```
/// Build request url
if (queryString)
  fullUrl += "?" + queryString;
```

**Listing 8.6. After**

```
/// Build request url (1)
fullUrl = buildRequestUrl(fullUrl,
      queryString);
/// ...
function buildRequestUrl(fullUrl: string,
      queryString: string) {
  if (queryString)
    fullUrl += "?" + queryString;
  return fullUrl;
}
```

(1) Comment is trivial now

People tend not to like such long method names. However, this is only an issue for methods we call often. Languages have a property where the words we use most often tend to be the shortest. The same should be true for our code base. This is also obvious, since we need less explanation for something we use all the time.

### 8.4.1 Planning comments

The way such comments most often come into being is by using comments to plan our work, and break down an elephant. This is a great way of creating a road map. I personally always plan out my code with comments like these:

**Listing 8.7. Planning comment**

```
/// Fetch data
/// Check something
///   Transform
/// Else
///   Submit
```

Some of these comments are likely to become trivial once the code is implemented, eg. `Else`. The others will be turned into methods. Whether you decide to turn these into methods up-front, is a matter of preference, but what is important is that once the code is written, we critically evaluate whether they add value.

## 8.5 Comments that document a non-local invariant

The final type of comments are those which document some non-local invariant. As we have discussed multiple times, these are where bugs tend to occur. A way to detect these is to ask "Will this comment ever prevent someone from introducing a bug?"

When we encounter these we still want to check whether we can make them into code somehow. In some cases we can eliminate these comments with the compiler as described in the last chapter. However, this is rare, so our next thought should be whether we can make an automated test to verify this invariant. If both of these turns out to be unfeasible, we keep the comment.

In the following example we see a suspicious statement `auth.logout` accompanied by a comment explaining the reason for the statement. Testing authentication, or complex

interactions like these can be dreadfully difficult to test or simulate, and therefore the comment is perfectly justified.

**Listing 8.8. Comment documenting an invariant**

```
/// Log off use to force re-authentication on next request
session.logout();
```

### 8.5.1 Invariants in the process

That something is undone (todo), is (probably) erroneous (fixme), or is a work-around of some third party software (hack) are all invariants. Not invariants in the code, but invariants of the process. Some people despise these, and argue justly that they should not be in the code but in our ticket system. I agree that this is also perfectly value, although I do prefer the locality of comments directly in the code. If they are in the code though there should be some visual indication of how many there are, and this number better be going down. We should stride towards actually fixing or doing the thing they are noting, so we can remove them, rather than postponing it further.

> *The best time to plant a tree was 20 years ago. The second best time is now*
>
> — Chinese proverb

## 8.6 Summary

- Comments can be useful during development, but we should try to remove them before we deliver.
- There are five types of comments.
- Out-dated comments should be deleted, as they can cause bugs.
- Out-commented code should be deleted, since the code is already in version control.
- Trivial comments should be deleted, because they do not add any readability.
- Comments that could be a method name, should be a method name.
- Comments that document a non-local invariant, should either be turned into code, an automatic test, otherwise we keep the comment.

# 9 Love Deleting Code

***This chapter covers:***

- Understanding how code slows down development.
- Setting limits to prevent accidental waste.
- Handling transitions with the strangler fig pattern.
- Minimizing waste with the spike and stabilize pattern.
- Deleting anything that does not pull its weight.

Our systems are useful due to the functionality they provide. The functionality comes from code, therefore it is easy to think that code is implicitly valuable. This is not the case code is a liability. It is a necessary evil we have to live with to get functionality.

Another reason that we tend to feel code is valuable, is that it is expensive to produce. It requires skilled workers to spend lots of time, and caffeine. Attributing value to something because we have spent time or effort on it is called the *sunk cost fallacy*. Value never comes from investment alone, but from the outcome of the investment. This is crucial to understand when working with code since we have to continually put effort into maintaining the code no matter if it is valuable or not.

Every programmer has tried getting bored with a manual task and thinking "I can automate this." In many cases, this is indeed why we became programmers. However, we have also tried getting distracted with the automation code so much that it steals focus from the original problem, and ended up spending more time automating it than it would have taken to solve the problem manually.

Writing code is fun, and it exercises our creativity and problem-solving skills. But the code itself is an expense for as long as we keep it around. To get the best of both

worlds we can do katas and spikes throughout our careers for training, and experimenting in code that is immediately deleted afterward.

In 1998 Christopher K. Hsee did a study called *Less is Better*. In the study, he established the value of a 24 piece dinner set. He then added a few broken pieces in addition to the original, and what he found is that the value decreased! Even though he only added it diminished the value. We need some long-living code in our systems, how much varies depending on the underlying complexity of the domain. However, if you take one thing from this chapter it should be: Less is better.

In this chapter, we look first at how we get into trouble with problematic code, through technical ignorance, waste, debt, or drag. We then dive into several specific types of code that we impose a drag on development, such as version control branches, documentation, or features themselves. We then discuss how to either overcome the drag, or get rid of it.

## 9.1 Deleting code may be the next frontier

Programming has gone through many phases. Here is a brief chronology of some of the big leaps programming has taken.

**1944** **Computers were used to perform calculations without any abstractions.**

**1952** **Grace Hopper invented the first linker, allowing computers to work with symbols instead of pure calculations.**

**1957** **The next step was the invention of the compiler, we could now code using high-level control operators like loops.**

**1972** **The next big issue that we tried to solve was data abstractions, enter the next generation of languages. Programming languages like C — and later C++ and Java — work with data indirectly, through pointers.**

**1994** **Another big leap forward was the invention of reusable design patterns.**

**1999** **Then came refactoring.**

**2011** **The most recent big leap forward in programming in my opinion is microservices architectures. Microservices architecture is based on the old principle of loose coupling, but they solved a modern scaling issue.**

We are now proficient at writing code and building systems. The systems we can build now are so big and complex that no person can reasonably understand them fully. This makes it challenging to remove things because to figure out what can be removed we need to invest time figuring out what code is being run, how often, in which versions.

We are not yet excellent at deleting code. I believe that this could be the next big problem to be solved.

## 9.2 Delete code to get rid of technical-ignorance, -waste, -debt and -drag

It is the nature of systems in use to grow over time as we add features, do experiments, and handle more corner cases. When we implement something we need to build a mental model of how the system behaves, and then make a change to affect that. A bigger codebase means a more complex model, because of couplings and a larger library of utilities to keep track of.

This complexity comes in two types: domain complexity and incidental complexity. Domain complexity comes from the underlying domain. That is, the problem we are solving is inherently complicated, eg. a system for calculating taxes will be complicated no matter what we do because the tax law is complicated. Incidental complexity is any complexity that is not demanded by the domain, but got added incidentally.

Commonly incidental complexity is synonymized with technical debt. However, in my experience, there are four types of incidental complexity, each with a different origin, and each has a different solution. Therefore, I think it is beneficial to have more finely grained terms. I categorize incidental complexity in four subcategories: technical ignorance, technical waste, technical debt, and technical drag. Let's discuss each in turn.

### 9.2.1 Technical ignorance from inexperience

The simplest type of incidental complexity is *technical ignorance*. This comes from unknowingly making bad decisions in the code resulting in poor architecture. This happens when we lack sufficient skills to solve a problem without adding unnecessary coupling, either because we don't know we don't know, or because we don't have time to learn. Hopefully, this book has helped reduce this situation for you. The only sustainable solution to this challenge can be found as the first half of one of the principles in the manifesto for agile software development:

> *Continuous attention to technical excellence and good design enhances agility.*
>
> — Manifesto for Agile Software Development

We must all continuously strive towards getting better at our craft, through reading books and blog posts, watching conferences and tutorials, knowledge sharing through pair- and mob programming, and most importantly deliberate practice — nothing is a substitute for practice.

### 9.2.2 Technical waste from time pressure

The simplest type of incidental complexity is *technical waste*. This comes from making

bad decisions in the code resulting in poor architecture.

Much more commonly technical waste stems from some form of time pressure. When we don't understand the problem or the model well enough, and are too busy to figure it out. When we skip testing or refactoring because we don't have time. When we circumvent a process to hit a deadline.

These bad decisions are intentional. In all cases the developer chooses to go against better knowledge, albeit due to external pressure. This is sabotage.

***A story from the real world***

Once, I was tech lead on a project, where we had slowly introduced a set of practices to ensure we didn't repeat mistakes of the past. We were under a lot of time pressure for the next delivery, so I asked one of my developers whether function X could be done by tomorrow. He replied: "Yes if I can skip testing." Biting my tongue, I told him that "done" meant following *all* our practices.

The solution is to teach developers that there is absolutely no occasion for skipping best practices. Teach project managers, customers, and other stakeholders that building software right is essential. I do this by asking them something like whether they would want their car three weeks earlier if it meant the brakes or airbags wouldn't have been tested? Some industries have regulations, developers have practices, we have to stick to them, even when pressured.

### 9.2.3 Technical debt from circumstances

Whereas both technical ignorance and waste can and should be eliminated, technical debt is more nuanced. *Technical debt* is when we temporarily choose a suboptimal solution, for some gain. This is also a deliberate decision, but the keyword being temporary. If this is not temporary it is not debt, it is waste.

One example where this happens often is when we implement a hotfix, without any regard for proper architecture, and push it to fix a critical issue. And then having to start over implementing a proper fix afterward.

I want to underline, this is a strategic decision, and there is nothing inherently wrong with it, as long as it has an expiry date.

### 9.2.4 Technical drag from growing

The final type of incidental complexity is the fuzziest. *Technical drag* is anything that makes development slower. This includes all the other categories, but also includes documentation, tests, and indeed all code.

Automated tests (intentionally) make it harder to change code as we also need to change the tests. Notice this is not necessarily bad, eg. for critical systems where we usually prefer being slow and stable over being fast. The opposite is true in situations that benefit from a high level of experimentation, such as during a spike.

Documentation slows us down because we need to update it when we change

something. Indeed even the code itself is technical drag because we need to consider how changes will affect the rest of the application, and we need to spend time maintaining it.

Technical drag is a side effect of building something. It is not in itself bad, but it is in situations where we are maintaining documentation, features, or code that are being used only sparsely. In such cases, it may be economically beneficial to simply remove the feature entirely, to get rid of the drag.

***A story from real life***

Once, as a developer on a project, I was asked to build a specific subsystem. I did so, but when it was done the customer was not ready to adopt it yet. The tech lead told me to just leave it in there, so it was ready for when they were. From that day, in everything we built we had to take into account how new code would react *if* they would suddenly start using this new subsystem. Of course, they never did.

The all too common argument "it doesn't hurt anything to keep it in there" is false. The solution is to delete as much as possible, but no more. Anything that is not paying for itself should go, even if it is being used a little. Delete every unused or unnecessary: feature, piece of code, documentation, wiki page, test, configuration flag, interface, version control branch, etc.

> *Use it or lose it.*
>
> — Proverb

Having established that everything incurs a drag, a slowdown on development, we spend the remainder of the chapter going into detail with the most common situations where we can get rid of things without losing value.

## 9.3 Categories for code based on intimacy

Before we dive into deleting specific things we need to take a detour and discuss three categories of code based on intimacy. In a conference talk Dan North categorizes code based on three levels of intimacy. We are intimately familiar with the code we have recently developed. We are familiar with the libraries and utilities we use often. Everything in between is unknown, and therefore expensive to maintain because we need to relearn it.

Relating to technical drag, the code we are familiar with because we use it often can stay. It also underlines the point that using things often is the only way to prevent them from decaying into the unknown. But it also adds a time component. Deleting code we are intimately familiar with is cheaper than code we have to first understand.

Dan North argues anecdotally that after about six weeks the intimacy of fresh code starts to quickly deteriorate, as code moves into the unknown category. The specific time is not important to me. It is important to keep in mind that there is a cutoff when

the author no longer has any meaningful head start for understanding code. And my experience agrees with Dan North's that it should be on the scale of weeks rather than months, so when referring to this later in the chapter I assume six weeks.

## 9.4 Deleting code in a legacy system

A common definition of legacy code is: code that we are afraid to modify. This is often the result of a circus factor. The circus factor[1] expresses how many people need to run off and join a circus before so much knowledge is lost as to halt some part of development. If we hear a statement like "Only John knows how to deploy this system" we say the circus factor is one in that system.

We never want to stop development, so we want to minimize risk by keeping the circus factor high. However, even if everyone on a team knows about all the code, sometimes a whole team is let go, or taken over from/by consultants.

When we lose our circus factor we inherit unknown code which we are likely reluctant to touch; legacy code.

The code may be working, but the fact alone that we do not feel comfortable editing it is enough of a negative that we should fix it. We need to be comfortable without code, and we need to take responsibility for it, to be productive. This cannot happen if it is fragile or unknown.

Having some part of the code be dark also means we have no idea when or how it can break, and worse: who should fix it when it brakes at 3 am on a Saturday?

### 9.4.1 Using strangler fig pattern to get insight

The first step in the solution is to find out how much it is being used. If it is barely being used at all it might be possible to simply remove it without further investigation. If only a small part is being used much we may only need to fix that part and get rid of the rest. Or if all of it is being heavily used we need to get comfortable with it, and possibly make it stable.

Getting insight into legacy code we need to know how much each part is being called. But this is not enough, we also need to know how many of these calls are successful, because some code is called, but failing so the result is never used, this is especially common in legacy code. Finally, we need to know how tightly coupled the legacy code is to the rest of the software. I recommend starting with the latter.

We can use Martin Fowler's *strangler fig* pattern to help with all of this. Named after strangler figs which seed on an existing tree and while growing envelope and ultimately strangle their host. In this metaphor, the host is the legacy system. The pattern proceeds as follows.

**Listing 9. 1. Legacy code**

```
class LegacyA {
  static a() { ... }
```

```
}
class LegacyB {
  b() { ... }
}

LegacyA.a();
let b = new LegacyB();
b.b();
```

To find out how tightly coupled some code is we can isolate it, making all accesses go through a virtual gate. We do this by encapsulating the classes in a new package/namespace, we then make a new gate-class inside the new package. We reduce all public modifiers inside the new package to be package-private, and going through the errors outside we fix them by adding a public function in the gate-class.

**Listing 9. 2. Before**

```
class LegacyA {
  static a() { ... }
}
class LegacyB {
  b() { ... }
}

LegacyA.a();
let b = new LegacyB();
b.b();
```

**Listing 9. 3. After**

```
namespace Legacy {
  class LegacyA {
    static a() { ... }
  }
  class LegacyB {
    b() { ... }
  }
  export class Gate {
    a() { return LegacyA.a(); }
    bClass() { return new LegacyB(); }
  }
}

let gate = new Legacy.Gate();
gate.a();
let b = gate.bClass();
b.b();
```

At this point, we know exactly how many contact points the legacy code has because they are all functions in the gate-class. We also have an easy way to add monitoring now, since we can simply put this in the gate class, we log every call and whether it was successful or not. This is just the bare minimum, and can be made as sophisticated as desired.

**Listing 9. 4. Before**

```
namespace Legacy {
  // ...
  export class Gate {
    a() { return LegacyA.a(); }
    bClass() { return new LegacyB(); }
  }
}
```

**Listing 9. 5. After adding monitoring**

```
namespace Legacy {
  // ...
  export class Gate {
    a() {
      try {
        let result = LegacyA.a();
        Logger.log("a success");
        return result;
      } catch (e) {
        Logger.log("a fail");
        throw e;
      }
    }
    bClass() {
      try {
        let result = new LegacyB();
        Logger.log("bClass success");
        return result;
      } catch (e) {
        Logger.log("bClass fail");
        throw e;
      }
    }
  }
}
```

We put this in production and wait. The team has to decide how long, but I don't think it is unreasonable to say that we don't maintain features that are not being used at least once in a month. Certain things have scheduled uses such as quarterly, biannual, or annual financial reports, but these I don't consider exempt from the treatment below. After this has been in production for a while we know how much each part of the legacy code is being used, and whether some calls always fail.

### 9.4.2 Using strangler fig pattern to improve the code

How much something is being called is usually a good indicator of how critical it would be if it failed. I like to start with the easy decisions, the most called ones should almost certainly be migrated, and the lest called ones can almost certainly be deleted, so I handle the extremities first, and move towards the middle, where the hard decisions are. For the code that wound up being called the least or always failing, we should critically assess whether it is critical or strategic functionality.

If some legacy code is critical or strategic we should first make sure the call number reflects this. We can increase the number of calls to the functionality through improving the UI, training, or marketing. Once the usage reflects the importance we need to get comfortable with the code. We have two options for this, either refactor that part of the legacy code, thereby removing coupling, fragility, and moving the code into the "recent"-category. Alternatively, we can rebuild the part, which we can easily switch to by changing the gate once the new code is ready.

If some legacy code is not critical and not strategic delete the method in the gate. This can sometimes make large parts of the legacy code unused, which we can find out with IDE support for methods, and ***Try delete then compile*** [P4.5.1] for interface methods. This also simplifies the calling code, as we remove a coupling, sometimes making this code deleteable as well.

Figure 9.1 summarizes how to deal with legacy code.

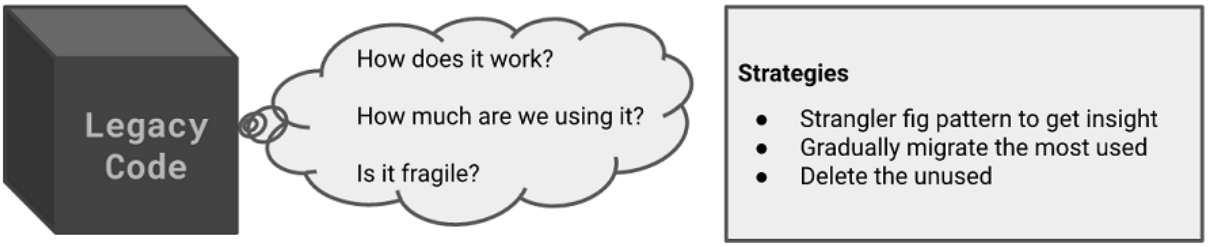


**Figure 9. 1. How to deal with legacy code**

## 9.5 Deleting code from a frozen project

Sometimes a product stakeholder requests a major feature. We start working on it, but by the time we finish there is some barrier: getting the necessary access, the users are not trained, etc. Instead of wasting time waiting we move on and work on the next thing. But now we have a *frozen project*.

Frozen projects don't need to be only code, they can include database tables, integrations, services, a whole host of things external to the code. These mean that once the original author forgets about the project it can be near impossible to spot that it is. Especially if what was missing was only the user training, there is no trace of this anywhere in the system, so no investigation will discover this.

We have code possibly on the master branch, that is not being used. There is no indication in the code that it is not being used, which means we have to consider it whenever we make changes, and we have to maintain it. This both adds to the mental overhead, and easily risks becoming legacy code.

Another problem with these is that there is no guarantee that the functionality will still be relevant when the barrier is removed.

### 9.5.1 Make the desired outcome the default

Depending on whether the project is exclusively in the code, or has effects on databases, services, integration, etc. there are slightly different solutions. We take each in turn.

If the project has no effects outside of the codebase we can revert the project off the master branch, and put it in a separate branch. Then we need to tag it, and make a note six weeks in the future to delete the tag. This means that if we don't take it into use within six weeks, it will be removed.

If the project includes changes external to the code we cannot necessarily put it in a branch. Instead, we should make a ticket in our project management tool noting all the components to remove and schedule it for six weeks. If this happens frequently it might be beneficial to make scripts for setting up and tearing down the most frequently used types of components.

In both cases, you will notice that unless deliberate action is taken the code will disappear. Therefore, in these scenarios, you cannot accidentally add technical drag, only deliberately.

### 9.5.2 Minimize waste with spike and stabilize

Another way to save effort when we have to implement a major change is by using Dan North's *spike and stabilize* pattern, wherefrom the six-week rule originally comes. In this pattern we treat the project as a spike, meaning we implement it as separate from the regular application as possible, and with no attention to high quality, ie. no automated testing, no refactoring. But crucially we put some monitoring so we can know how much the code is being used.

After six weeks we return to the code, and see if there has been any usage. If there has, we reimplement it but the right way, with refactoring and everything. If not we delete it, which is easy because the spike had already minimal integration with the main system. So we save time on removing it, but we also save the time we would have spent refactoring or testing the code without knowing whether it would ever be used.

Figure 9.2 summarizes how to deal with frozen projects.

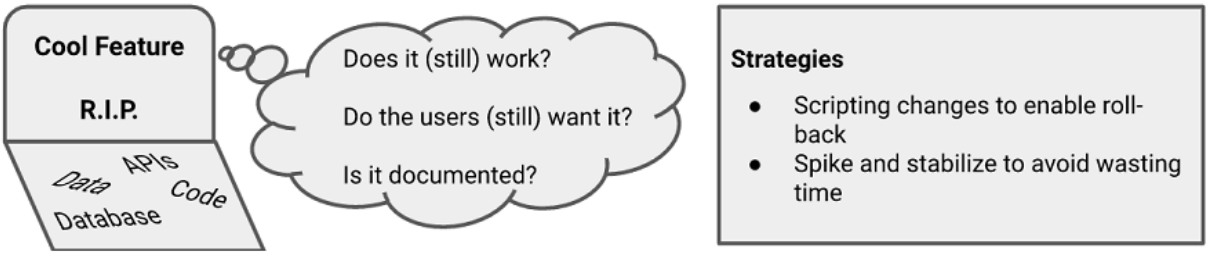


**Figure 9. 2. How to deal with frozen projects**

## 9.6 Deleting branches in version control

Branches behave differently in different version control systems. In centralized version control systems like subversion, they duplicate the entire code base, making them quite expensive. Whereas in Git branches require mere bytes, independent of the size of the codebase. In this section, we consider only Git branches because the issue tends to resolve itself if branches are expensive.

When branches are cheap we tend to be less diligent about removing them, thus they build up over time. We create branches for a lot of purposes; The main reasons fall into these categories:

1. To do a hotfix.
2. To tag commits, we may need to return to later like a release.
3. To work on something without interfering with our colleagues' work.

Categories 1 and 3 should be deleted once we merge into master. In 2 we should instead use Gits built-in method for tagging; tags. Knowing this why do they build up? Sometimes it is simply an oversight, like when we forget to tick the “delete branch” when merging our pull requests, or forget to remove an experiment branch after we are done. Sometimes they host frozen projects, spikes, or prototypes, because what if we think we might need the code someday.

These are relatively easy to deal with a more difficult type of branches are those that are pending, but blocked because they cannot pass the gate to get onto master. This happens if our gate includes a human component like an integration team, or asynchronous human code review. Both of these prevent continuous integration, and easily become a bottleneck, slowing down development. But if they only cost a few bytes what is the harm of leaving them?

Like code, branches in Git are technically almost free, but expensive in mental overhead. We should only have a master branch, and possibly a release branch, but any other branch should optimally live only days. Having long lived branches we expose ourselves to expensive, soul-crushing, and error-prone merge conflicts. And clutter causes more clutter.

### 9.6.1 Minimize waste by enforcing a branch limit

To solve this issue we can adapt an element from the development method Kanban. Kanban uses a concept of Work In Progress (WIP) Limits, which means we have a set ceiling on how many tickets the team can have in progress. This limit helps expose bottlenecks in our development because a bottleneck will eventually hit the WIP limit, which in turn prevents people upstream from starting new work. Not being able to start new tickets people upstream are encouraged to investigate what the bottleneck is, and how to resolve the clog. This encourages teamwork, and continuous improvement of the process.

This issue exactly mirrors the issue described above, so we can reuse the same solution. By introducing a hard limit on the number of branches. Let’s go through a few things to keep in mind when setting a WIP or branch limit. The limit should at least be equal to the number of workstations so everyone can work in parallel, a workstation here is a unit that can work independently, such as one mob if we are doing mob programming, one pair if we are using pair programming, or one developer otherwise. Setting the limit higher has the effect of building a buffer, which imposes a delay in the system but can be useful if some work tends to be significantly varied in size. We desire as little delay in the system as possible. Most crucially, once a limit is set it should not be broken or changed for any reason short of changing team size.

Figure 9.3 summarizes how to deal with branches in version control.

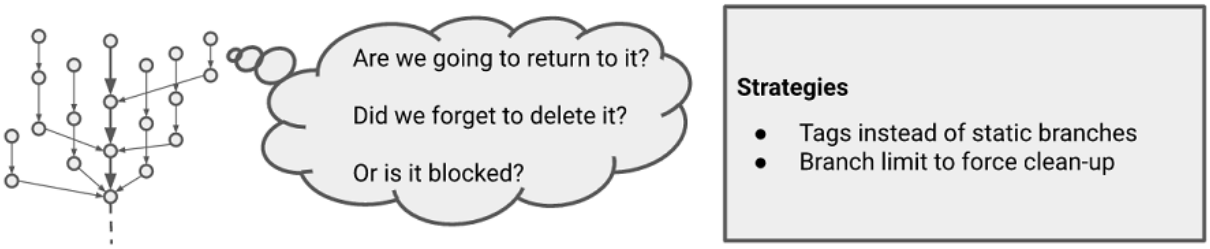


**Figure 9. 3. How to deal with branches**

## 9.7 Deleting code documentation

Code documentation comes in many forms: wiki pages, javadoc, design documents, tutorials, etc. As we dealt with intra-method comments in the last chapter we do not consider these here.

Documentation that is invaluable when exactly three conditions are met:

- relevant — it needs to answer the right question,
- accurate — the answer needs to be correct, and
- discoverable — we need to be able to find the answer.

If any of these are missing the value of the documentation is greatly diminishes. Writing good documentation is difficult, and requires effort to make sure it stays relevant and accurate. This is because documentation needs to be used at least as often as the subject changes, otherwise maintaining it will likely not be cost beneficial. Keeping it up to date can happen through frequent adjustments, or by generalizing it upfront abstracting away parts that change often.

The danger of keeping outdated documentation around depends on which of the three properties it violates. The least significant is if the documentation is not discoverable, then only the writing time is wasted and the research time. Worse is keeping around irrelevant documentation because in this case writing time is wasted, but we also need to skim past it every time we are looking for answers, and in the end, we still have to do the research. Worst is inaccurate documentation which can in the best case cause confusion and doubt, but in the worst case can cause errors.

### 9.7.1 Algorithm to determine how to codify knowledge

Because documentation can lose the properties above not everything makes sense to document. It might seem like documentation saves you from having to repeat previous research, but that is only the case if the documentation does not drift out of date. When I need to determine whether it makes sense to document something I go through this process:

- if the subject changes often then there is nothing to be gained
- else if we will use it rarely then document it
- else if we can automate it then automate it

- else learn it by heart

Notice that a solution can be to increase the usage of some documentation, causing the frequent adjustments mentioned above. This can be by making new team members go through it, and correct anything inaccurate. Doing so tends to require some confidence to determine whether the documentation is wrong or the person did something wrong, so when in doubt the person should simply flag the difference.

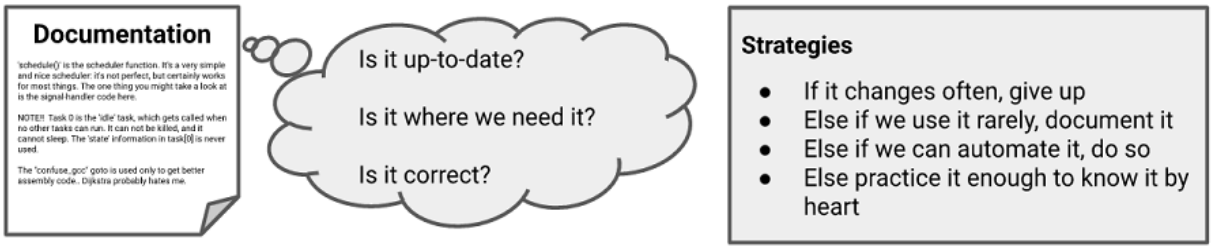


**Figure 9. 4. How to deal with documentation**

Figure 9.4 summarizes how to deal with documentation. Another approach to documentation that stays accurate is to use automated test cases as documentation, so let's examine that.

## 9.8 Deleting testing code

Automated tests (simply tests in this section) come in many flavors, and have a lot more properties than documentation. Kent Beck describes 12 properties of tests in his "Test Desiderata". Properties that different types of tests put different weights on. I will not go through all of that here, but instead focus only on tests that hurt development.

### 9.8.1 Delete optimistic tests

Sometimes we write some code, like a `hash` function, we want to test it, so we come up with a test that says: Given a = b then hash(a) = hash(b). This seems like something we want to be true. But we have accidentally stumbled upon a tautology; something that is always true.

One necessary property of tests is that they inspire confidence. A green test should make us more confident that something is working. This means they should test something, a test that cannot fail is worthless.

A nice concept from the test-first community is: "Never trust a test you have not seen fail." This is useful when we discover an error in our code, making a test before fixing the issue we can check that it correctly fail, whereas if we make it after we only ever see it pass.

### 9.8.2 Delete pessimistic tests

Similarly, a red test should mean that something is broken and we need to fix it. That is

we the tolerance for failing tests should be zero. If we have tests that are always red we risk getting alarm fatigue, and miss a critical error, even when the tests catch it.

### 9.8.3 Fix or delete flaky tests

Both optimistic and pessimistic tests are extremes, passing always or failing always. But the same issues apply to tests that are unpredictably red or green, sometimes called *flaky tests*. Like both types discussed above these also do not elicit any action, except perhaps running the tests a few more times. We act if and only if a test is red, any test for which this is not true has no place in our code base.

### 9.8.4 Refactor the code to get rid of complicated tests

A wholely different category are those tests that require delicate setup, or exhibit a lot of duplication, so we decide to refactor them, or build complicated test setups. These are dangerous because they feel like we are doing valuable work, we are simplifying, localizing, doing all the right things. Unfortunately doing the right things in the wrong place is still wrong. If the test is more complicated than the code, how do we know whether the code is wrong or the test? Even when this is not the case, the need to refactor tests is a sign that the code being tested does not have proper architecture, and any refactoring effort should be in the code not in the test.

### 9.8.5 Specialize tests to speed them up

In some places, we use end-to-end tests to check that certain functionality works. This certainly has its place, but these tests can be slow, and having many will impact how often we can run the tests. If some tests are causing us to run other tests less often they are hurting development, and we need to address this. There are two ways to do this one is to separate the slow tests from the fast, and keep running the fast tests as often as possible. The other solution is to observe what causes the slow test to fail, if the answer is nothing it is an optimistic test and should be removed. Likely there are only a few things deeper in the system that tends to go wrong in which case we can make tests for those places. These tests will be faster, and more specific, this means we can correct errors faster.

Figure 9.5 summarizes how to deal with automated tests.

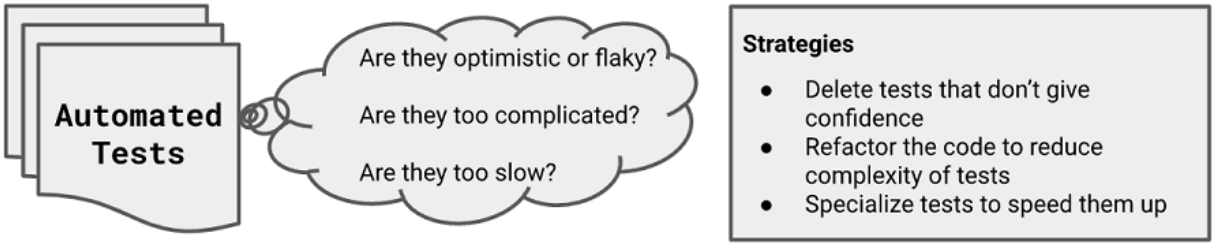


**Figure 9. 5. How to deal with automated tests**

## 9.9 Deleting configuration code

That hard coding is bad is known to most programmers. The first solution we learn to deal with this is to extract the hard coded value into a constant. Then as we mature we learn the maxim:

> *If you can't make it perfect, at least make it configurable.*
>
> — Maxim

Configurability can increase the utility of our software when it means we can increase our user count, without a significant increase to the code base.

When it comes in the form of feature flags it let's us separate deploy and release, increase deployment frequency, and make release a business decision instead of a technical decision.

It does however come with a price, as each place we add configurability we increase the complexity of the code. Even worse in most cases, we double the testing space, because we need to test for each option against all other flags. This grows exponentially. Hopefully, some of the flags are independent and can be tested at the same time. Taking advantage of this makes it possible to test it however, we do open ourselves up to errors involving complex interactions between those flags.

### 9.9.1 Scoping configuration in time

My solution to dealing with the increase in complexity due to configuration is to consider as much of the configuration temporary as possible. To this end, I categorize configuration based on its expected life-time. The categories I suggest are: Experimental, transitional, and permanent.

#### EXPERIMENTAL CONFIGURATION

We have already talked about an example of experimental configuration: feature flags. These are intended to be removed after the release of the feature, and to ensure this is an easy task it should be done within the six weeks discussed earlier. Another type of experimental configuration comes from testing whether a change is superior. This is sometimes called beta-testing, or AB-testing. In the code, these are very similar, but their purposes differ. In this case, the configuration is there to allow some users to experience the change while some do not. This way we can gauge whether a change will have the desired effect, ultimately we are looking to determine whether before or after is superior. This allows us to adjust to feedback, or opt-out of a change without affecting all users.

In my experience testing configuration tends to leak out of the experimental phase and become permanent, splitting the user base into those with the flag on, and those with it off, increasing only complexity and not usage. This is bad, so to avoid this we should be proactive; determine from the beginning whether something is experimental and upon completion create a reminder for its removal immediately after testing is finished,

keeping within the six weeks.

### TRANSITIONAL CONFIGURATION

The transitional configuration is useful while the business or code base is going through major changes. An example of this could be moving from a legacy system to a new one. We cannot expect or enforce that making large scale changes like this happens within 6 weeks, so we have to deal with a longer-term increase in complexity, and a higher clean-up cost. However, longer transitions like these usually have two properties that we can take advantage of.

First, the transition of many things is invisible to the user. Therefore we can be satisfied with release and deployment to be linked. This means we can have the configuration as part of the code, instead of something external. Having it in code means we can collect all the configuration tied to the transition it in a central spot, separated from the rest, which makes the invariant explicit that these are more closely related than other configuration flags, and should be considered as a collection.

Second, there is usually a point where the transition is complete and the old part can be removed entirely. Taking advantage of this we can avoid spending time chipping it up, and deleting small parts, and simply wait for the whole thing to go all at once. To make this safe we should again use the stranger fig pattern to gate *all* access to the legacy component. Not only can this work as an excellent todo-list, but when we can delete the gate without getting errors in our code, we know that we can delete the whole legacy component as well. We can either discover this through ***Try delete then compile*** [P4.5.1], or by gradually deleting methods in the gate as they become unused, and once it is empty we can delete it.

### PERMANENT CONFIGURATION

The final category is permanent configuration. This is special because it should cause an increase in usage, or be trivial to maintain. An example of usage-increasing configuration is reusing most of the same software for two different customers by putting their differences behind an in-code configuration flag. Or it could be configuration to enable different tiers of usage, allowing us to cater to different business sizes. Both of these could potentially double our number of users, making the configuration well worth an increase in maintainability.

An example in the trivial-to-maintain subcategory is offering a light-vs-dark mode flag to the users, this affects only the outermost parts of our code, the styling, and therefore does not affect maintainability, but it can enhance the experience for some users.

We should be very critical of what goes in the permanent category, and if it is not in one of these two categories it is probably not worth the cost and should be removed.

Figure 9.6 summarizes how to deal with configuration code.

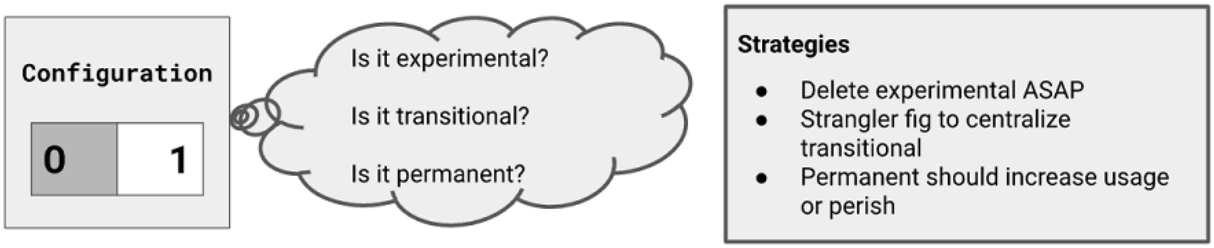


**Figure 9. 6. How to deal with configuration code**

## 9.10 Deleting code to get rid of libraries

A quick way to get a lot of functionality cheaply is using 3rd party libraries. Some libraries both spare you from writing thousands of lines of code, but also provide you with higher quality or security than you could have gotten in-house. I have always advised: leave security to people who devote their lives to it because us lay-people are simply not experienced enough to put up a fight against attackers, who often also devote their lives to their craft.

Another property of security that supports going with a 3rd party library is that the quality of our security can easily have a direct impact on our software's viability. If our software has a major security incident it may well destroy our users' trust, and thereby the software.

Another reason to use a 3rd party library is when it enables doing something that is not feasible without, such as using a front-end framework like Swing (Java), React (Typescript), or WPF (C#). These all provide a lot of code, which required specialized skills to build, graphics programming skills that we may not have in our team.

Unfortunately, using libraries is a double-edged sword, because although we don't need to maintain their code, we do need to update them, which sometimes means we have to adapt our code. This can be time-consuming, and error-prone. Using libraries also adds to the finite cognitive load of the team, because they need to keep at least a working knowledge of it.

We lose some predictability when we use libraries, as we cannot predict when updates are coming, or how much time we need to spend adjusting our codebase. Sometimes features we rely on are deprecated or removed and we need to build something to replace it. Sometimes bugs are introduced and we need to implement temporary workarounds or hacks to make our software work. Finally, when bugs are fixed in the libraries we need to undo our workarounds, so they don't fester in the code.

We are also forced to make a decision between either needing to read and understand the library source code, or accept degrading security, because the library is another possible attack vector, that we can only vouch for by treating it as we do our own code.

***A famous thought experiment***

In a blog post, David Gilbertson presents a thought-provoking fictitious scenario where he releases a small JavaScript library for adding colors to log messages in the console. He reports: "People love pretty colors" and "we live in an age where people install npm packages like they're popping pain killers." With minimal social engineering (some pull-requests) he injects his library into other libraries. The library starts getting hundreds of thousands of downloads monthly. However, unbeknownst to the users, the library contained malicious code that steals data from sites using it.

The danger of external libraries is amplified because most modern languages come with a package manager, which makes it easier than ever to add dependencies. And as the scenario above illustrates, we need only worry about our dependencies, but also all the dependencies of our dependencies, and so on.

### 9.10.1 Limiting our reliance on external libraries

One method for dealing with the pains described above is picking libraries from high-quality vendors. Where we trust their internal quality and security requirement. Venders who strive to avoid breaking changes. We only need to re-audit for security, or adjust our code when there are updates, so if the libraries change rarely we are minimizing these costs.

Another way to reduce the pains is to update frequently. In DevOps there is a saying:

> *If something hurts, do it more.*
>
> — DevOps proverb

If we do something often we have more incentive to streamline it, and reduce the pains it causes. This argument is behind processes such as continuous integration and delivery. Another advantage to doing something more often is that the amount of work tends to be smaller, spreading out the cost and reducing the risk and cost overall.

This does however not help mitigate the security risk mentioned in the previous section. The final and simplest solution I will suggest is: Make your dependencies visible, then categorize whether each library is *enhancing* or is *critical*. Use this to lower your dependence, and ultimately reduce your reliance on libraries.

If an enhancing library breaks simply remove it, get the application working, then look for a replacement later. Be cautious about promoting a library from enhancing to critical. If we have unused libraries lurking in our code base remove them. If they are relatively easy to implement in-house that is often worth it, to remove the uncertainty.

If we have installed the jQuery library, with its hundreds of functions, but we are only using the one to make ajax calls, it would probably be advantageous to either find a simpler library that fits our needs more precisely, or even implement our function for this. In terms of security, we need to audit all the code in the library, even if we don't use it directly.

Figure 9.7 summarizes how to deal with 3rd party libraries.

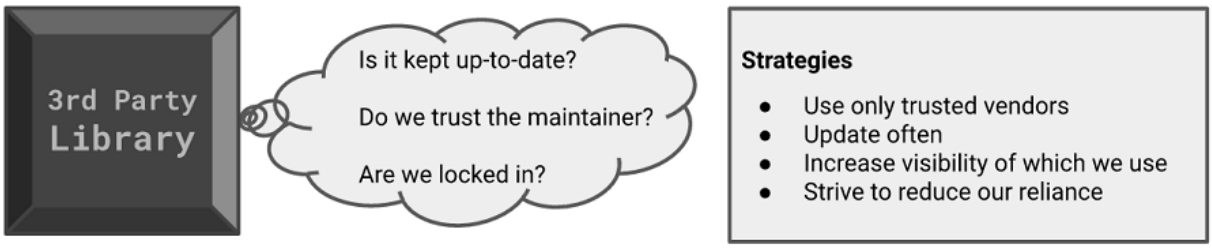


**Figure 9. 7. How to deal with 3rd party libraries**

## 9.11 Deleting code from working features

Code is a liability, it cost time to maintain it has a lot of unpredictability and therefore comes with risk. Usage is the value, that pays for it. It is a common misunderstanding that features are correlated with usage so that adding more features adds more value. Unfortunately, it is not so simple.

As I have tried to illuminate throughout this chapter there are many factors at play when balancing the cost of code with the benefit of functionality. How long we accept an increase in complexity, how we value predictability, how we test new features, how well we onboard people to them. There are two ways to increase value in any cost/benefit relationship, and given that the benefit of features is so complicated, it is often easier to look for ways to reduce the cost, by refactoring, or even better: removing code. This is even true if you remove working features whose cost is bigger than the usage increase they cause.

Analogous, anything that is unused, no matter its potential is simply only an expense. This is why you should love deleting code because it immediately makes the code base more valuable.

Figure 9.8 summarizes how to deal with working features.

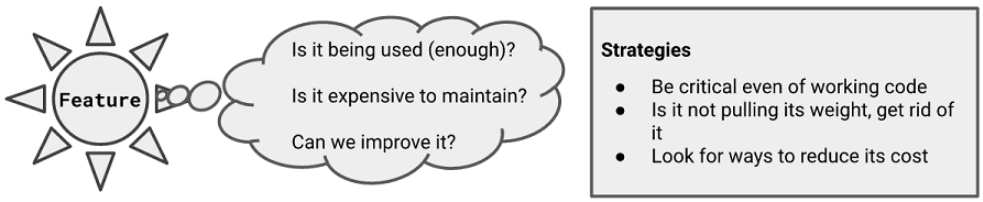


**Figure 9. 8. How to deal with working features**

## 9.12 Summary

- Technical-ignorance, -waste, -debt and -drag express different reasons why development gets more difficult and slower. Technical ignorance usually stems from inexperience and is handled only by a continual focus on technical

excellence. Technical waste often comes from time pressure, but since it provides no benefit, is it only sabotage. Technical debt arises from circumstances, and is perfectly acceptable as long as it is temporary. Technical drag is simply a side-effect of the code base growing, it is a necessity because our software models a complex world.

- We can use the strangler fig pattern to both get insight and delete code from a legacy system, or to centralize configuration during a transition period.
- By using the spike and stabilize pattern we can reduce some of the waste that comes from a frozen project. Further by making the default action deleting the project as opposed to keeping it, we prevent it from becoming drag.
- By deleting bad automated tests we increase confidence in them, and thereby making the test suite more useful. Bad tests include optimistic, pessimistic, and flaky. We can also improve the test suite by refactoring the code to get rid of complicated tests, and specializing slow tests to make them faster.
- By enforcing a branch limit we can reduce the cognitive load wasted on keeping track of stale branches in version control.
- By setting and keeping strict time limits on configuration we keep complexity creep to a minimum.
- By limiting our reliance on external libraries we save time on updating and auditing while increasing predictability.
- For code documentation to be useful it needs to be relevant, accurate, and discoverable. We discussed an algorithm to determine how to codify knowledge.

[1]. Also sometimes called the bus or lottery factor with more morbid or lucky metaphors respectively