

Data-Oriented Programming in Java

Chris Kiehl



MANNING



MEAP Edition
Manning Early Access Program

Data-Oriented Programming in Java

Version 11

Copyright 2026 Manning Publications

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

welcome

Thanks for purchasing the MEAP for *Data-Oriented Programming in Java*!

This book is a distillation of everything I've learned about what effective development looks like in Java. It's what's left over after years of experimenting, getting things wrong (often catastrophically), and slowly having anything resembling "devotion to a single paradigm" beat out of me by the great humbling filter that is reality.

Data Orientation is not some new paradigm here to beat up all other paradigms and take their lunch. If you like object orientation, functional programming, or any other paradigm, a little touch of data-orientation will make you better at those styles of programming! Data Orientation is about the *data*, not the specific tools. There are, of course, some patterns and approaches that naturally emerge when you focus on the data, but all of those can be applied readily to whatever paradigm is your preferred one.

This is because data-orientation is born from a very simple idea, and one that people have been rediscovering over and over again since the dawn of computing: "representation is essence of programming". Programs that are organized around the data they manage tend to be simpler, smaller, and significantly easier understand. When we do a really good job of capturing the data in our domain, the rest of the system tends to fall into place in a way which can feel like it's writing itself.

So, this book is about data. What it is, how to think about it, how to understand its semantics, how to model it, how to represent it in our code, and, somewhat surprisingly, how to listen to its feedback. The act of trying to capture what data *is* can often reveal how much we don't understand about the domain we're supposed to be modeling. To quote Bertrand Russel, "everything is vague to a degree you don't realize until you try to make it precise." Data-Orientation gives us the tools for making things precise.

All you'll need to follow along is a basic working knowledge of Java. As long as you know what a class is, and how to define an interface, and have at superficial understanding of generics (i.e. you've used a type like `List<String>` before), you've got everything you need to follow along.

Thanks again for purchasing this book! Please share your thoughts and questions in the [liveBook Discussion forum](#). Tell your friends and coworkers to buy a copy, too. I need the royalties to buy a yacht.

—Chris Kiehl

Brief contents

PART 1: FOUNDATIONS

[1 Data Oriented Programming](#)

[2 Data, Identity, and Values](#)

[3 Data and Meaning](#)

[4 Representation is the Essence of Programming](#)

PART 2: DOMAIN BEHAVIORS

[5 Modelling Domain Behaviors](#)

[6 Implementing the Domain Model](#)

[7 Guiding the design with properties](#)

[8 Business Rules as Data](#)

[9 Refactoring towards Data](#)

[10 Data Oriented Architecture](#)

[11 Testing Data Oriented Programs](#)

1 Data Oriented Programming

This chapter covers

- Introducing Data-Oriented programming
- Data as Data
- How representation effects on our programs

This book is about data. What it is, how to think about it, how to model it, how to represent it in our code, and all the good things that happen when we do. Programs that are organized around the data that they manage tend to be simpler, smaller, and, most importantly, significantly easier to understand.

We're going to learn how to model data "as data" using Java. Meaning data on its own, as ordinary values, independent of any class, operation, or behavior. We'll still use those things throughout our programs, but at the heart of everything will be data, and its representation independent of any other code.

We lift data up to this lofty place because it *means* something within its domain and to the people that use it. Data is more than just a collection of values that gets shoveled around our programs. It's more than that stuff inside of our objects. Data has a semantics which differentiates it from all the other data it *could* be. An integer can represent an infinite number of "somethings", but it's only within a particular domain that it becomes something meaningful like an age or population count. It's by breaking out data on its own that we can focus on this semantic meaning. When we understand our data at a deep level, we unlock new expressive power in our programs.

At its core, data-oriented programming is ultimately just learning to be really, really precise about what we mean. That's pretty much the whole trick. Studying the data on its own enables us to move away from ambiguous generalities and towards explicit representations that capture the essence of what we're modeling. Before we get to the question of "what does it do?", we try to capture something much more fundamental, bordering on philosophical, the question of "What is it?". If we can answer that question and express the data's meaning in our code, some amazing properties start to emerge which can reshape our programs.

1.1 Objects in a Data Oriented World

Since we're all Java programmers, it's worth clearing this up before we go any further: Data-orientation doesn't mean giving up objects! Data orientation is not some new paradigm here to replace all the others and shame you for ever having used them. Objects are invaluable tools. I've tried programming without them in other languages and I always come crawling back. It's tough to beat the object's ability to effortlessly manage stateful runtime resources (thread pools, socket connections, resource lifecycles, etc.). Since objects aren't the enemy, the only mental shift required in the data-oriented approach is to view objects not as "The One True Way" to build software, but as something much more humble: another tool available for our use. Objects are great at some things and less good at others. This is true of all things that are tools. A

hammer is not good at delicately tightening a watch screw. It *is* very good at being a hammer, though. It excels at all hammer related tasks.

So, the main shift in data-oriented programming is about *where* and *how much* we use objects. We put them where they excel: up at the top of our applications where they can enforce boundaries (encapsulation boundaries, maintenance boundaries, “past this line, I don’t want to know or care about X” boundaries). One of the object’s strengths is that it naturally pushes us towards creating these strong isolation boundaries in our code. It puts a subtle design pressure on us that nudges us towards higher level abstractions. This is great when we’re drawing up the borders in our application as a whole. The pressure nudges us in the right direction. However, this design pressure can get in the way when we’re dealing with certain kinds of business logic. Sometimes what our programs really need is not another layer of abstraction or indirection, but to drop down a level of power, and work directly with the concrete, plain, easy to understand data sitting at the bottom of everything.

Listing 1.1 A class modeling a piece of data

```
class Point {  
    private final double x;    #A  
    private final double y;    #A  
    #B  
}
```

#A Our data is just the attributes which make it up. This class doesn't do anything other than just be information

#B (getters, equals, and hashCode skipped for brevity)

This can feel a bit funny at first. Maybe even bordering on “wrong.” We generally don’t approach modeling data this way in Java (though we happily use it when it exists). The atomic unit of object orientation is data *and its behaviors* together, coordinating as one. Further, good object-oriented design encourages us to abstract “above” the data as much as possible, and focus on the interfaces through which our objects interact. The object-oriented design process is largely the act of figuring out where you draw borders, and who calls which interface, and which piece is responsible for what (along with other concerns of and making sure each object has enough to do, but also not too much). For all the guff Java gets for being the “kingdom of nouns,” we actually spend the bulk of our effort stressing about how the verbs attached to those nouns interact with one another.

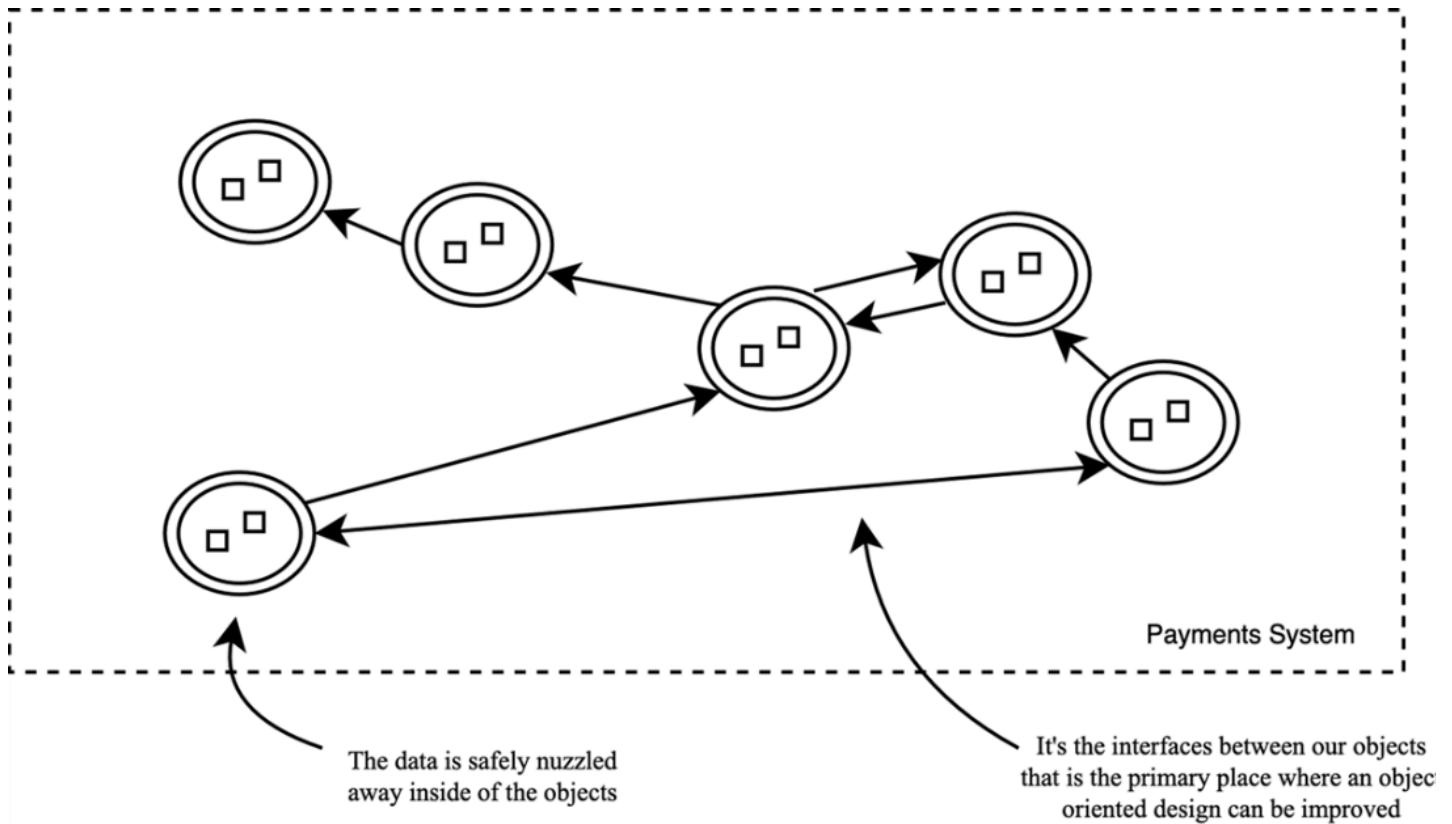


Figure 1.1 Objects and how they communicate is our focus during object-oriented design

The process looks much different when we focus on “data as data” during our design phase. The *representation* of that data becomes the primary point of interest (it also becomes the main point where we can improve a design). We get to take a brief step back from the complexities of objects and how they interact, and instead focus on the much simpler question of “what is this stuff?”. We spend a lot of our time exploring what the information in this domain is and understanding the semantics which govern it at a fundamental level.

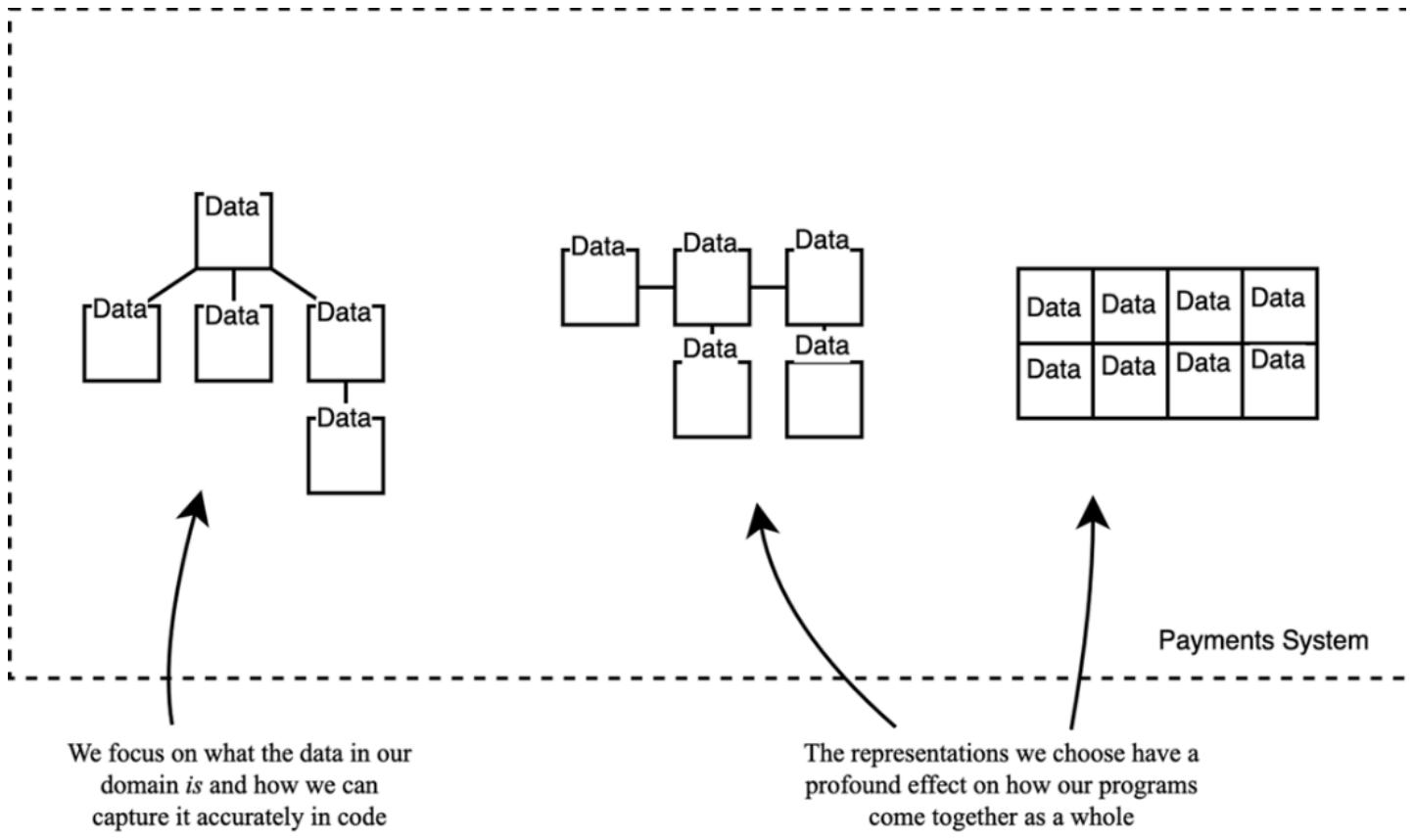


Figure 1.2 The representation of our data is the primary focus during data oriented design

So where do we put the objects? When do they come back into play? My sell to you is that if we do a really good job of representing our data, the objects we need to support it tend to naturally emerge in the right spot (often in a way that can feel almost inevitable). They seem to fall naturally into place because their role, vending a piece of data we've already designed, gets figured out during our modeling phase. All that's left is gluing everything together atop our strong foundation of data.

It will definitely take some getting used to. There will be some challenges ahead of us. However, if we can quiet that little voice of discomfort and skepticism long enough to get over the initial hump, there's an exciting style of programming to explore. Focusing on modeling data as data puts a very different kind of design pressure on us from the one we usually feel under object orientation. It nudges in a direction that makes us view our programs under a new light. When data is out there on its own, it suddenly needs to do something that it never needs to do when encapsulated behind an object: describe itself. Without the encapsulation of a class to give it context though behaviors, it has to communicate what it is through other mechanisms. We're forced to consider what the data really means and, most importantly, how it should be represented. It is our data's "representation", to quote Fred Brooks, that "is the essence of programming." When we get it right, everything else tends to fall into place.

1.2 The soul of data-oriented programming in a single line:

To start to explore the outsized effect that data's representation has on our programs as a whole, we only need a single line. We'll give ourselves a sole, individual piece of data.

An identifier of some kind. It'll be out there on its own. Unadorned and without any kind of containing object.

Listing 1.2 One line. A vague identifier of some kind

```
String id; #A
```

#A What would be right value to provide here?

So, the question is, without a class to contextualize it, just what *is* this thing? What does the current representation of this data communicate to us as readers of the code?

The answer I'd give is "not much." When we encounter this code, all we know is that it involves a variable called "id," and that it's a String, and... that's it. The representation doesn't tell us anything about what the id is supposed to be. The information that's in the code barely narrows it down at all. A string can represent just about anything. So, the thing we have to answer is, of that just about anything that it *could* be, what *should* it be? What is a meaningful ID for our domain? Figuring this out isn't always straight forward. More often than not, this domain information doesn't live anywhere in the code itself. Instead, we have to leave the code entirely to chase down the information elsewhere -- either a more tenured coworker, external docs or wikis, or (on (frustratingly many) projects I've worked on) watching production traffic to see what is flowing across the wires.

And this is the core of the problem: communication. The current representation doesn't communicate anything to us about what this piece of data is supposed to *be*. Even if we give a better name than `id`, or add a bunch of comments and java doc, the problem remains that the code itself -- the stuff we program with -- doesn't communicate anything about the semantics of the data. It doesn't tell us what kind of thing this identifier is meant to be.

The current representation is a little speed bump in the code that slows down each person's understanding of what's going on. It's small and trivial in this example, but small ambiguities grow into incomprehensible ones over the course of a project. This is an ambiguity that people will have to spend time resolving. If they do it right, the only cost is some time. If they do it wrong, the cost is usually a bug.

So, here's the data-oriented view of this: when we're designing our data type, which, in this case, represents an identifier of some kind, we'd look at that ID and go: "Ok. What does it really mean to be an ID in our domain?" It's *probably* something with more specific semantics than that String is conveying. Another way of thinking about it would be to imagine if we were to drop someone fresh into this part of the code. What is it that we'd want them to know about this id? How can we communicate that knowledge to them in the code directly?

As for our identifier, it could be anything, but let's say, for ease of example, that IDs are actually UUIDs in our particular domain.

Now we know what it *is*, we can ask if the code captures the semantics of "being" a UUID. Our `String`, while it can *technically* hold a `UUID` in string form (and is extremely common to do so), can also hold all kinds of things that aren't a UUID.

Listing 1.3 One line. Many wrong answers. The wrong ways to create IDs.

```
String id = "not-a-valid-uuid"; #A
String id = "Hello World!"; #A
String id = "2024-05-04"; #A
String id = "172.16.24.105"; #A
String id = "1010011001011011"; #A
```

#A All things which are not UUIDs, yet allowed by our String type

Our one-line example here, while about as trivial as it gets, embodies the problems that an imprecise representation can cause in our programs. There's a mismatch between what we "know" something means ("This ID is a **UUID**") and what the code says it is ("literally any String is A-OK"). This creates an interesting dynamic in the code. If you just think about what's allowed to be expressed -- what kind of values *could* you create with this code, it's actually kind of dire. There's an infinite number of things that *aren't* UUIDs! And our representation allows any of them to be incorrectly applied to our `id`!

We usually don't try to quantify how many "wrong" states our program can enter for this reason. Facing down something like "an infinite number of wrong ways to do something" is a heavy burden for a programmer to bear. The "standard" approach to this problem might be to try to wrangle some of those illegal states under control by sprinkling in some defensive programming throughout the code wherever this data gets used. A precondition here, an `if` check there, maybe a full Anti-Corruption Validation Layer at the front door. Regardless of how we do it, we *have to do it somewhere*, because the representation allows those wrong states to be expressed, which means they could be entered (and on a long enough time scale: *will*). And mounting those defenses means writing more code. And then tests of that code. And then more code after that. All to work around the fact that the representation we've picked for something that's supposed to *only* be a UUID allows things that aren't UUIDs.

That brings us back to the data-oriented view. All of the problems, like the infinite number of wrong ways of creating an ID, and the need for defending against those not UUIDs, stem directly from how our data is *represented* in the code. A String is not a good way to represent a UUID. It allows too many things that aren't UUIDs. We know what our data is supposed to be, so we need to bring the representation of the data in closer alignment with it.

In the case of our UUID, this is super simple, because Java has a ready-made type we can use. We can swap out the ambiguous String for the concrete UUID.

Listing 1.4 Changing how we represent that one line

```
String id
UUID id #A
```

#A Swapping a generic and ambiguous String for an explicit UUID

Which is an obvious change, right? ("obvious" things that aren't common currency are going to be a very big theme of this book). However, what's interesting about changing this one single line, as "obvious" as it might be, is that it fundamentally changes what the code communicates to us as readers. We've made the code *describe* itself. There's no ambiguity. We don't have to chase down coworkers or dig through databases. What it means to be an `id` in our domain (A UUID) is expressed directly in the code.

This subtle shift in representation, despite being a single line, has a massive impact on our program as a whole. We've moved the semantics of what it means to be an ID out of our heads and into where it belongs: the code. As a result, something kind of magical happened: all of the illegal "not-UUID" states disappeared! Actually something even more profound than that happened. Our program can now *only* construct correct states. There is literally no way to create anything that isn't a valid UUID. As such, there are no bad states to defend against, so there's no need for additional code. And no need for tests of that additional code. It can be hard to quantify because it's invisible, but picking a better representation made our *entire* code base smaller and simpler because of all the *other* code we now don't have to write.

There's one more thing that's really interesting about this little "obvious" example: how infrequently we collectively take this incremental step towards making our code express itself. At the time of this writing, if you search for "String id" on Github, you'll find 4.5 *million* instances of this exact ambiguity in Java alone. Some of those might very well actually mean "an ID is literally any conceivable String," however, I suspect that most of those represent a gap in the developer's modelling -- an instance where a just a *little bit* of unnecessary friction gets placed on the programmer's ability to understand the code. Data Orientation is largely just taking that tiny incremental extra step towards the "obvious" representation.

And that's the basic idea in a one-line nutshell. Good representation ripples outward throughout our codebase. It has a simplifying effect because of all the other code we don't have to write. It eliminates illegal states and, in doing so, makes our programs as a whole smaller, simpler, and easier to reason about.

Let's keep going with a bigger example. Data is everywhere in our programs, but it's often hidden just below the surface due to the flexibility that encapsulation allows. A big part of data-oriented programming is training ourselves to notice that data – to lift it up to the surface so that we can make it explicit. The more we clarify the data in our domain, the more understandable our programs become.

1.3 Show me your data, and the rest will be obvious

Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious

Fred Brooks

One of the central ambiguities that pops up in most mainline code stems from being unclear about what the state inside of our objects means. We assume our readers will just "get it" while reading our code in the same way we do while writing it. However, so often, there's a mismatch between what we *think* we've expressed in the code, versus what we've *actually* expressed in the code.

This presents as explanations that go "well, if this one field is set, then it means *this*, but if this *other* field is set then it means..." (often times this keeps going "if they're both *not* set then it means..." (and going "but when we set...")). The state inside our objects is being used to model... something, but that "something" is only alluded to and hinted at. It's never expressed directly. It's something that works fine when we're the

people writing the code and have the implicit meaning tacitly in our head. However, it becomes impenetrable when we're the ones piecing together why that one method sets that one field some of the time.

Here's an example of what I'm talking about. It's a small slice of a larger set of classes that deals with running scheduled tasks asynchronously. We're going to give ourselves a simple job: understand what the `reschedule` method does. We're purposefully going in blind and ignoring anything else that might be in the class for now so that we can really focus on what the code communicates on its own, exactly for what's written on the page.

Listing 1.5 if attempts is *not* set then it means..."

```
class ScheduledTask {  
    private LocalDateTime scheduledAt; #A  
    private int attempts; #A  
  
    void reschedule() { #B  
        if (this.someSuperComplexCondition()) {  
            this.setScheduledAt(now().plusSeconds(this.delay()));  
            this.setAttempts(this.attempts() + 1);  
        } else if (this.someOtherComplexCondition()) {  
            this.setScheduledAt(this.standardInterval());  
            this.setAttempts(0);  
        } else {  
            this.setScheduledAt(null);  
            this.setAttempts(0);  
        }  
    }  
}
```

#A These two instance variable power everything this class does.

#B Our job is to try and understand this method and what it does with the variables

The code itself, we could all probably agree, is trivial. The statements are all easy to follow, and there aren't too many of them. A few conditions and a few variable assignments. However, easy to read doesn't always mean easy to understand! If you actually try to reason about what this code does, you'll find that something fundamental is actually left completely unsaid in the code: what does all of this *actually* do? I don't mean what fields does it set, that's obvious. What's *not* obvious here what it *means* when those fields are set.

The big challenge with understanding code like this is that you have to take off your engineering hat and swap it out for something more like a psychologist hat. We have to peer inside the dark recesses of the original developer's mind to attempt to piece together what they were thinking. Those fields *mean* something when they take on (or don't take on!) certain values. The author of this code had a model in their head, and they were using these particular values assigned to these particular fields to represent it. However, they did so only partially. They encoded just enough of their world to make their intentions clear to the computer, but we humans are left guessing.

Of course, it's not totally grim. Dealing with this type of ambiguity might as well be what programming *is* most of the time. It's what we do every day. If you stare at this code long enough you could start to make some decent guesses as to what the various

branches mean. For instance, in the first branch, where `someComplexCondition` is true, `attempts` gets incremented. However, that same `attempts` variable is reset in all other branches. Pair those together and you have interesting clues that there's some kind of lifecycle going on here. It's unstated in the code, but we can kind of see it in between the lines if we squint.

Listing 1.6 Looking closer at the reschedule method

```
void reschedule() {    #A
    if (this.someSuperComplexCondition()) {
        this.setScheduledAt(now().plusSeconds(this.delay()));
        this.setAttempts(this.getAttempts() + 1);    #A
    } else if (this.someOtherComplexCondition()) {
        this.setScheduledAt(this.standardInterval());
        this.setAttempts(0);    #B
    } else {
        this.setScheduledAt(null);
        this.setAttempts(0);    #B
    }
}
```

#A Interestingly, we can note that attempts is only incremented in the first branch

#B It's reset in all the others. What does that mean?

When it comes to understanding what these variable assignments mean, the absolute best we can do within the confines of this method is make a guess. All we have are some variable assignments. There's just not enough information to build up a working theory of why these values are set or what it means when they are. We've reached an informational dead end because the code doesn't communicate anything to us.

So, we have to leave the method and go foraging around in the wider world. The code doesn't tell us what these variable assignments mean, so we have to look at where and how they're used. It's only by studying everything that we can inductively start to build up our own mental model that maps meaning onto what these individual variable assignments denote when they hold certain values.

If you're lucky, you'll find usages elsewhere that give context as to what a particular assignment means. Here's an example we might find in another class that gives some meaning to one of the possible states.

Listing 1.7 finding clues throughout the code

```
class Scheduler {
    ...
    private void pruneTasks() {
        this.getTasks().removeIf((task) ->
            task.getScheduledAt() == null           #A
        );
    }
}
```

#A Aha! We find a critical clue to what these things mean. Whenever scheduledAt is null, it seems to mean "give up on this one" and it gets evicted from the scheduler.

The process is to just keep going like this, over and over, poking around the codebase, reading all of the implementations, until we've found a large enough set of examples to

construct our own model of what's going on in the code – what the code means.

Table 1.1 the meaning we've worked out for various states assigned in the reschedule method

When these are set	Then it seems to mean...
attempts+=1; scheduledAt = small delta	Go for an immediate retry
attempts=0; scheduledAt=large delta;	Give up for now and try again later
attempts=0; scheduledAt=null	Give up on this job entirely

This process always represents hard won knowledge. It could be a few minutes, hours, or even days depending on how complex the code is. The unsatisfying part of all of this is that you can never *really* be sure that your analysis and model of the world lines up with the one originally envisioned. We build up a theory by plugging each example we find into our own evolving inductive model of the world. However, there could be another usage lurking in the code that invalidates all of our assumptions. It's a similar problem to the old observation about the limits of software testing: you can only prove the existence of bugs, never their absence. Understanding an existing codebase written in this style presents us with the same limitations. It's a lossy process. Without a direct line to the original developer, we can only reason about their intention from the examples we see in the code. We can reason inductively about the examples we've seen, but we can never really be sure we're correct, because the code itself doesn't tell us.

This lack of explicit and clear modeling presents a very real problem. Beyond just being plain hard to understand, I'd argue that it's one of the most common sources of bugs in our programs. To use some lingo from the world of databases, the code in our example lacks "semantic integrity." Those variables mean something when they're set, and the interpretation of that meaning is critical to the correct functioning of the software, but that meaning isn't enforced or encoded anywhere. It's left entirely implicit, which means each and every part of the code has to reinterpret that implicit meaning over and over again. This dooms us to semantic drift as time goes on. All it takes is for one method to misinterpret what a particular state *means* in order for the whole thing to collapse in on itself.

Which all brings us back to the data-oriented view. There's some data in there that's floating just below the surface. It's the ideas behind what these individual variable assignments mean. It's not currently expressed, but we can bring it to light if we focus on it.

This is where representing data "as data" comes in. Focusing on just the data by itself lets us take a step back from the details of the code as a whole and instead just think about, ok, what is it that I'm *really* talking about? What does it *mean* when I set those fields? I've got some mental model floating around in my head, and it has a semantics that I'm implicitly honoring. How do I get that out of my head and into the code?

So, what are we really talking about in this case? When a ScheduledTask fails, our system has to make a decision about what to do next. What we've been piecing together as we explored this code is that it's not just any arbitrary decision, it selects

from a fixed set of next possible decisions. That's what's ultimately going on with those variable assignments. It's just hidden by the current modeling.

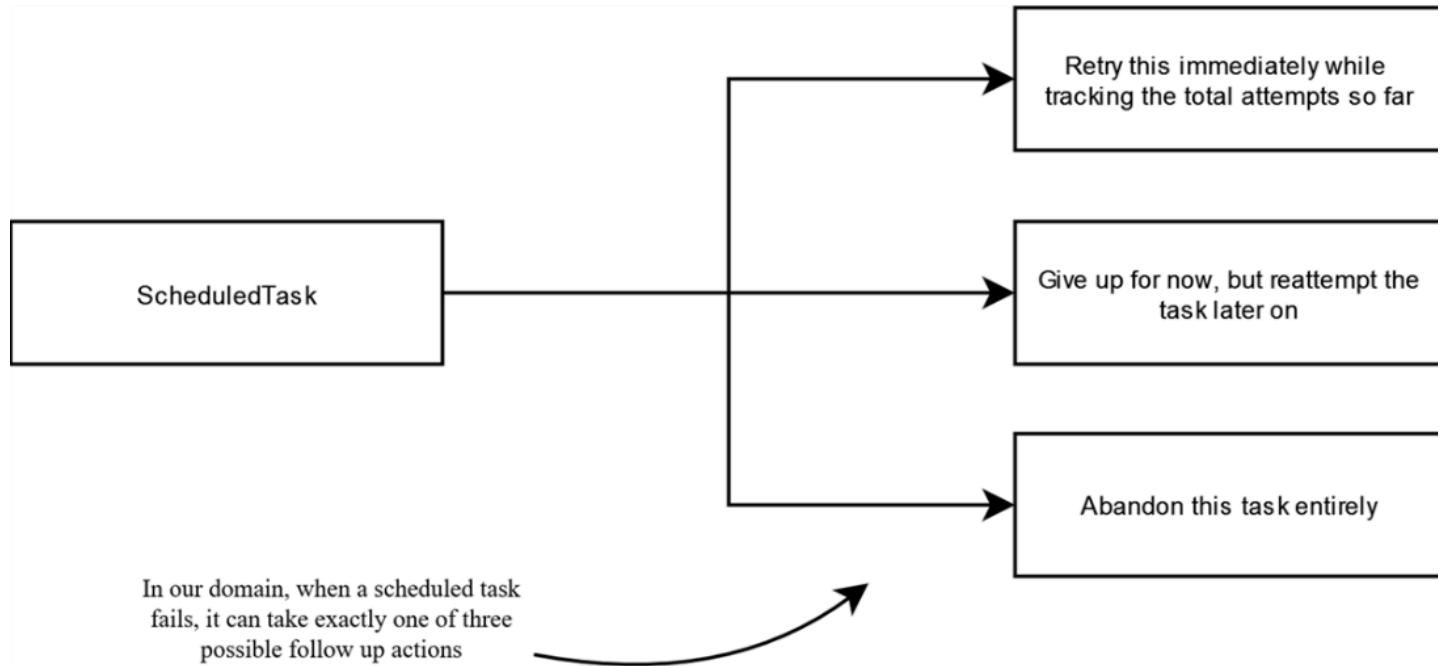


Figure 1.3 Being explicit about what a task can transition to after failing

So, let's start there. Let's model these decisions that our system can make as individual pieces of data. Modern Java gives us lots of options for how to do this, but for now we'll just use the humble class. But to be clear, even though we're using a class, our intention is not to create an *object*. We won't be defining any behaviors. These classes will be representing data. They'll be defined entirely by their name and their attributes. Nothing else. (Technically, these classes won't quite be "data" as we define it later in the book, but we'll hand wave that away for now. They'll be good enough for our current example. Exploring how we get objects to act like data is the topic of the next chapter).

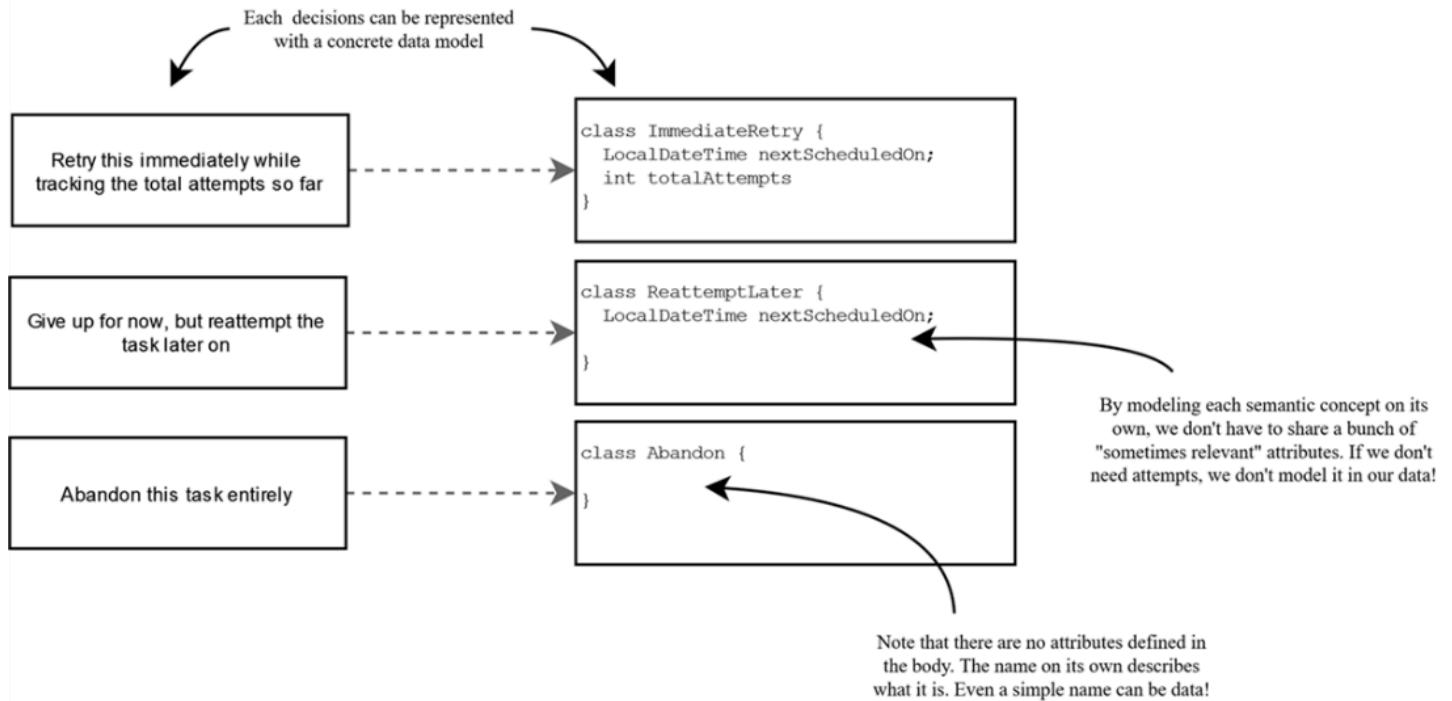


Figure 1.4 Representing each decision as a piece of standalone data

Look how descriptive those data types in Figure 1.4 are. They don't do anything – they're just data about a behavior that the system is allowed to do elsewhere. But despite just sitting there being data, they tell us so much about important ideas within our domain. We can understand what happens after a task fails just by looking at our data. We don't have to guess or interpret vague state assignments – it's obvious and explicit!

Let's plug it into the original code.

Listing 1.8 Refactoring to make the semantics explicit

```
class ScheduledTask {  
    private LocalDateTime scheduledAt;    #A  
    private int attempts;                 #A  
    private RetryDecision status;        #A  
  
    void reschedule() {    #A  
        if (this.someSuperComplexCondition()) {  
            this.setStatus(new RetryImmediately(      #B  
                now().plusSeconds(this.delay()),  
                this.attempts(status) + 1  
            ));  
        } else if (this.someOtherComplexCondition()) {  
            this.setStatus(  
                new ReattemptLater(this.standardInterval())  
            );  
        } else {  
            this.setStatus(new Abandon());  
        }  
    }  
}
```

#A We've swapped the vague instance variables for a new model

#B Our method now produces concrete, explicit information about what the decisions it's making mean

Don't worry about how we tie all these data types together for now (we'll get to that in later chapters). Instead, the thing we should pay attention to is the fact that by splitting out the data on its own and forcing it to carry its own weight, what the code as a whole communicates has been completely transformed.

This change is very similar to our one-line change in section 1.2. It's ultimately pretty small and obvious – maybe even underwhelming at first. However, its effects ripple throughout our codebase. You can now read this *one method* and understand a lot about the behavior of the system as a whole. We can understand something fundamental about the core ideas in our domain just from looking at the data types. No more guess work. No more hunting for clues. The code tells you exactly what it is. I mean, compare that to the variable assignments we had before.

Table 1.2 Comparing the meaning conveyed with different approaches

Old implicit variable assignments	Explicit concrete Data
attempts+=1; scheduledAt = small delta	class RetryImmediately { LocalDateTime scheduledAt; int attempts; };
attempts=0; scheduledAt=large delta;	class ReattemptLater { LocalDateTime scheduledAt; };
attempts=0; scheduledAt=null	class Abandon { };

And it doesn't stop there. By making our model explicit, we've baked semantic integrity directly into the code. That semantic meaning is available for use everywhere else in the code. We no longer have to look at a null and make a guess that it has some significant domain-level meaning. We get to work with those descriptive, explicit pieces of data:

Listing 1.9 No more guesswork about what variables mean

```
class Scheduler {  
    ...  
  
    private void pruneTasks() {  
        this.getTasks().remove((task) ->  
            task.getStatus() instanceof Abandon      #A  
        );  
    }  
}
```

#A Now relies on explicit semantics.

Between the two, which one conveys more information about the ideas in the domain?

Listing 1.10 comparing the two approaches

```
tasks.remove((task) -> task.getStatus() instanceof Abandon)  
// vs  
tasks.remove((task) -> task.getScheduledAt() == null)
```

One is explicit. One is not. In the data-oriented version, the code tells you exactly what it *is*. The author gives you a direct line to the model they had in their head.

WAIT -- ISN'T USING INSTANCEOF A TERRIBLE PRACTICE?

Seeing that `instanceof` in listing 1.10 might have set off some reflexive alarm bells in your head. Sprinkling `instanceof` throughout the code is generally a big "no-no" in the world of object-oriented programming. It's a strong smell that we're likely missing polymorphism somewhere.

You'll have to take it on faith for now (we'll explore the why in later chapters), but the difference here is that we're not dealing with objects, we're dealing with something representing *data*. It has a very different set of considerations and patterns from objects, and thus follows a different set of rules. That distinction won't be clear yet. So, for now, just mentally tuck this slightly suspect usage of `instanceof` away as "will be justified later"

WHAT ABOUT IF WE...

I can hear some of your potential push back over my examples so far. Depending on how you view those class boundaries, you might see the "real" problem with this code as the fact that `ScheduledTask` leaks internal information about its representation out to the world. Why should the `Scheduler` be peering inside of a `Task` to decide what to do anyway? A more object-oriented view might point us towards this being a shortcoming

in the *public interface* between these two classes. So, to be fair, we might adjust the original design by keeping its state the same, but exposing a more domain-level interface into our object. That stops us from leaking any internal details (and also gives us another example usage demonstrating what those internal states *mean*).

Listing 1.11 An alternative approach to clarifying meaning

```
class ScheduledTask {  
    private LocalDateTime scheduledAt;      #A  
    private int attempts;                  #A  
                                         #A  
    void reschedule() {...}             #A  
  
    public boolean isAbandoned() {        #B  
        return this.scheduledAt == null;   #B  
    }                                     #B  
}
```

#A Sticking with the original implementation for our state and reschedule method

#B But instead of leaking the inner details, we expose domain specific methods for the rest of the code to use

Like our data-oriented refactoring, this approach also has the same effect of rippling outward to clarify other parts of the code. For instance, here's what the prune method might look like using the new method.

Listing 1.12 Using a descriptive public method rather than internal state

```
class Scheduler {  
    ...  
  
    private void pruneTasks() {  
        this.getTasks().remove((task) ->  
            task.isAbandoned()           #A  
        );  
    }  
}
```

#A Calling a descriptive public method rather than looking at internal details

And this is great, too! It's a huge improvement from the original sketch we made. However, what I want to sell you on is that these two approaches are not at odds with each other. They complement and enhance each other! The main drawback of only focusing our efforts *exclusively* on the interfaces is that, well, it only makes those interfaces better. All those same semantic integrity problems exist *inside* the encapsulation boundary of our class. No matter how well named the things are that we expose, they're still powered by logic that requires us to spend time inside of our class reasoning through "well, if this field is set, then...". So, in our day-to-day work, we're still stuck playing detective in order to understand how the variables on the instances tie to the behaviors exposed to the world. And that's why we don't have to pick between one or the other – we can use both! A small touch of data orientation can make your objects better! In fact, a good representation can drive the design of your interfaces in a way that can feel like it's writing itself.

Listing 1.13 Combining Data Orientation with Object Orientation

```
class ScheduledTask {  
    private RetryDecision status;  
  
    void reschedule() {...}  
  
    public boolean isAbandoned() { #A  
        return this.status instanceof Abandon; #A  
    } #A  
  
}
```

#A Of course we'll have this method for the outside world to consume, it's a fundamental idea in our domain!

Combining objects with well modeled data is a great way to dip your toes into the world of data-oriented programming. It's a subtle shift from just thinking about objects as needing to hold *just enough state* to power their behaviors. Focusing on the data on its own can clarify what the stuff in our objects means.

1.4 Orienting around data

A few interesting things begin to happen as you focus on modeling more and more of your domain as data. A natural feedback loop occurs. The first is in the understanding of the domain itself. A particular kind of modeling pressure is placed on us when we're forced to make data describe exactly what it is, without any help from behaviors. It nudges us towards precision and a kind of "single responsibility," but not for what it *does*, but for what it *is* – what it semantically represents. The second thing is that how we construct our programs starts to be informed and influenced by the data itself.

For example, you might have noticed something kind of "off" when we were focusing on the data in our domain "as data" in the previous section. If we're just looking at what our data means and what it communicates on its own, it currently communicates something kind of strange. It says a scheduled task (whatever that looks like) is what is directly related to these decisions about what to do when we retry.

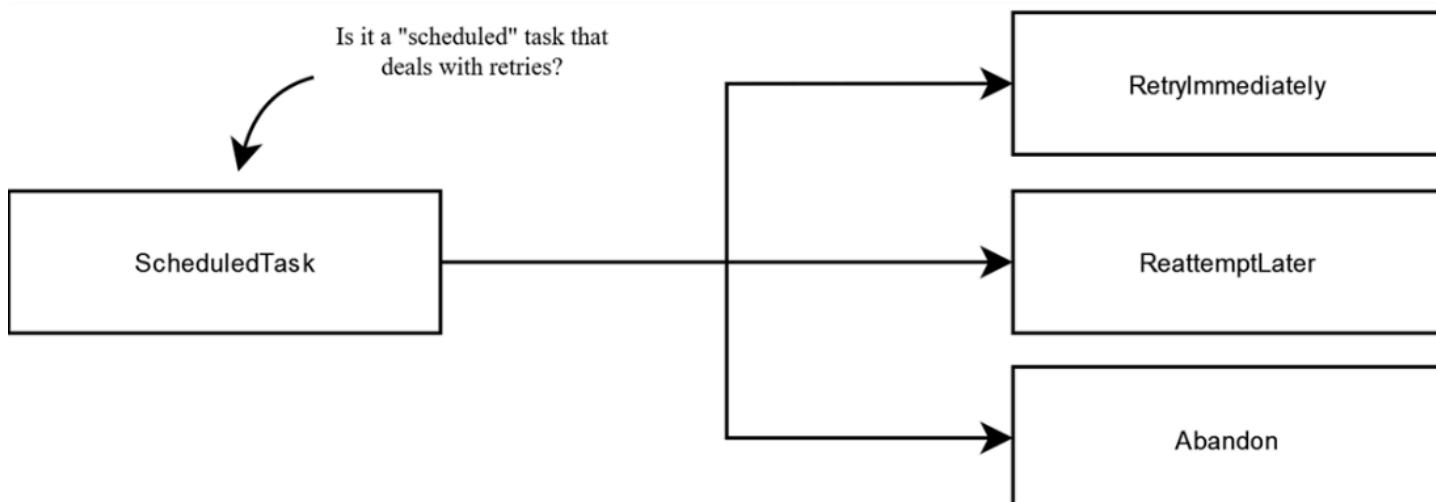


Figure 1.5 Focusing on just the data makes us question our representation

But that's not quite right, or, it doesn't capture the semantics of what's actually going on. It's too superficial. It's not a *scheduled* task needs to retry – it hasn't done anything yet! It's only a task that has run and failed for which these other pieces of data become relevant. Further, if we think about it, a failed task is probably going to have very different things it's worried about than a scheduled task (for instance, *why* it failed). There's this deeper level of semantics in our domain that we're forced to explore (and express!) so that our data "says" things that make sense.

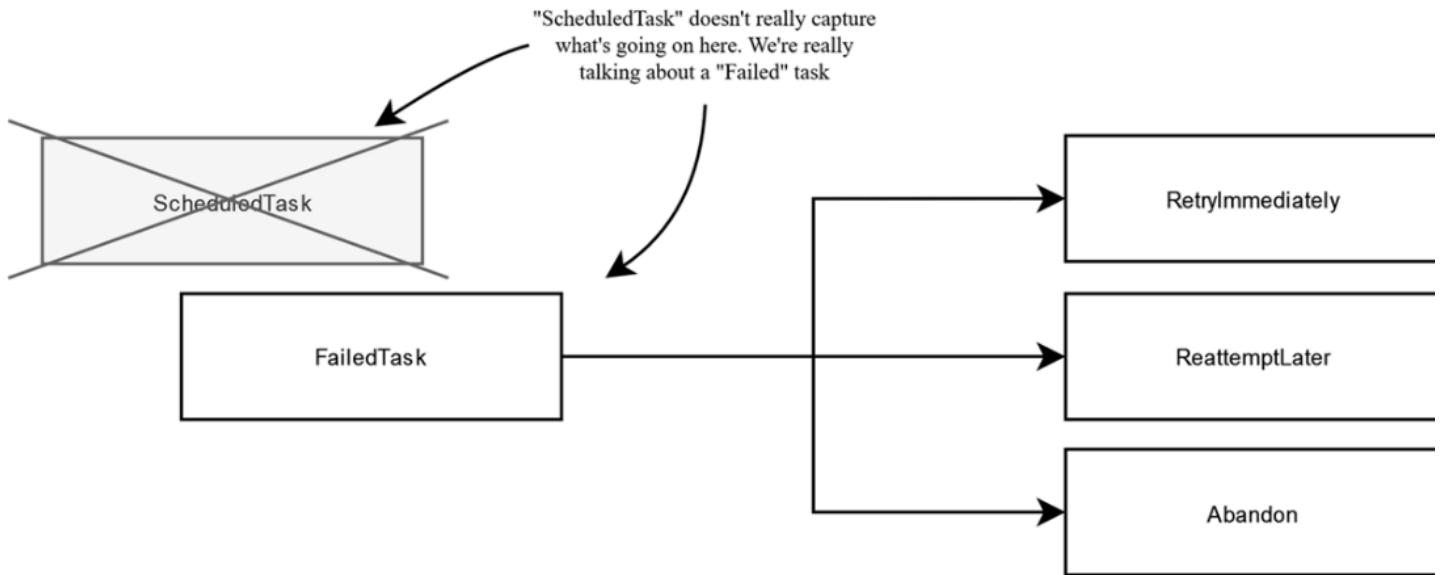


Figure 1.6 clarifying what we're talking about

This observation, which is usually a deeply satisfying "aha!" kind of moment during the design process, kicks off a feedback loop towards deeper insights. If we have this idea of a Failed Task, do we have other semantically unique states that need modeled? What about when a task doesn't fail? It probably has a result of some kind we need to track. Not to mention all kinds of specific meta data that only makes sense for tasks that have finished running. We could try to model all of these different ideas with an ever-growing number of instance variables that mean different things depending on where our task is in its lifecycle, but that'd put us back where we started: reasoning "between" the lines. Modeling data as data lets us precisely capture the ideas in our domain in a way that makes what they are obvious and clear.

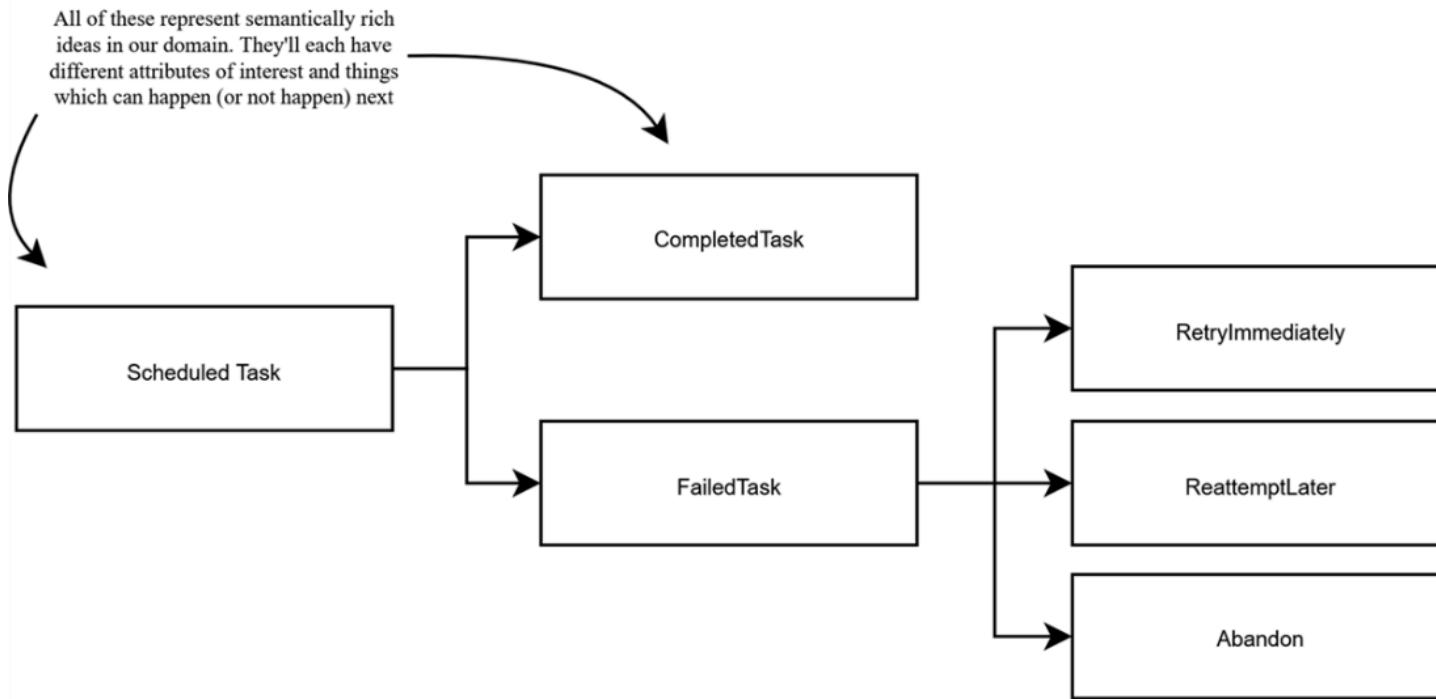


Figure 1.7 Analyzing the data drives a deeper exploration of the domain

And this brings us to how looking at the data as data can start to reshape our programs as a whole. How they will naturally start to “orient around” the data. It’s not something we focus on, it’s emergent from the data model. And it starts with our methods.

In listing 1.5, we looked at the body of the reschedule method and tried to figure out how much about our code as a whole we could understand just from its implementation. It started out as an opaque assignment of values to instance variables, then became explicit and clear once we made the transition to representing the intention of the method in plain data. However, there’s still something vague about the method as a whole. To understand what it does, we *have* to read its implementation. The *method signature* itself tells us nothing about what it means.

Listing 1.14 What does the method on its own tell us about what it does?

```

class ScheduledTask {
    private RescheduleDecision status;

    void reschedule() { #A
        // ...
    }
}

```

#A What can we know about what this method does without looking at its body?

We have this rich data model, but the code doesn’t take advantage of it. It buries it down inside of the implementation. This slows us down from a “what does this code actually *do*?” perspective. As we read through this class, we force ourselves to start from scratch on every method. When we hit that reschedule method, all we know is that it’s a void method that does... *something*. We have to look inside of it to find the good stuff. That’s when we realize, oh, there’s this whole world hidden behind that void. It’s hiding this notion of “failed” and that we’re ultimately making a decision about what to do when a task is in that state. We could speed up this understanding, and make the

code describe itself at a higher level, by moving that explicit, self-describing data into the method signatures themselves.

Listing 1.15 Which tells you more about what the method does?

```
void reschedule(){} #A  
// versus  
  
static RetryDecision reschedule(FailedTask task) { #B  
    // ...  
}
```

#A This method takes nothing and returns nothing. Because of that, it tells us nothing.

#B Contrast that with this one. We don't have to read the body to know what this does – it tells us! Its behavior is obvious from the data that goes into the method, and the data that comes out.

This is *expressive* code. It tells us exactly what it is. No guess work. No hunting around the codebase. What you see is what you get. It's clear. It's boring. It's immediately understandable.

Data will also reshape how we design the internals of our methods. When their main job becomes taking data as input and returning data as output, they tend to take on a more expression-oriented shape. Meaning, rather than a series of imperative statements and assignments (like we had in listing 1.5), they are built out of expressions that return values.

Listing 1.16 Refactoring the implementation to take data and return data

```
public RetryDecision rescheduleTask(FailedTask task) {  
    return switch(task) { #A  
        case FailedTask t when someSuperComplexCondition(task)  
            -> new RetryImmediately(delay(), t.attempts() + 1);  
        case FailedTask t when someOtherComplexCondition(task)  
            -> new ReattemptLater(this.period());  
        default -> new Abandon();  
    };  
}
```

#A Using Java's new switch expression to enumerate the decisions

These effects naturally flow outward from simply listening to what the data has to say. It can feel like the code as a whole starts to write itself. For instance, what should the run method of the scheduler look like? If we're only thinking about interfaces, this is something we have to "design". However, because we've thought through the data, it's already "designed" for us – we just follow where the data leads. It takes and returns the relevant data types.

Listing 1.17 Comparing type signatures

```
public void run(){...} #A  
  
public Result<CompletedTask | FailedTask> run(ScheduledTask) { #B  
    ...  
}
```

#A A void method that does... something (where we might have ended up without focusing on the data)
#B Versus the lifecycle of a task on full display (Note! we're showing the two return types for ease of example, but this isn't valid Java! We'll explore the actual technique for returning multiples data types in later chapters)

This is what I mean by objects naturally falling into place. The data nudges us towards structuring our methods around inputs and outputs. Because of this, the inside of our objects tends to look more like a series of pipelines. They take data in a particular shape on one side, and produce data in another shape on the other. Objects still communicate like objects, but they speak in terms of data. Their job becomes retrieving, managing, and vending data.

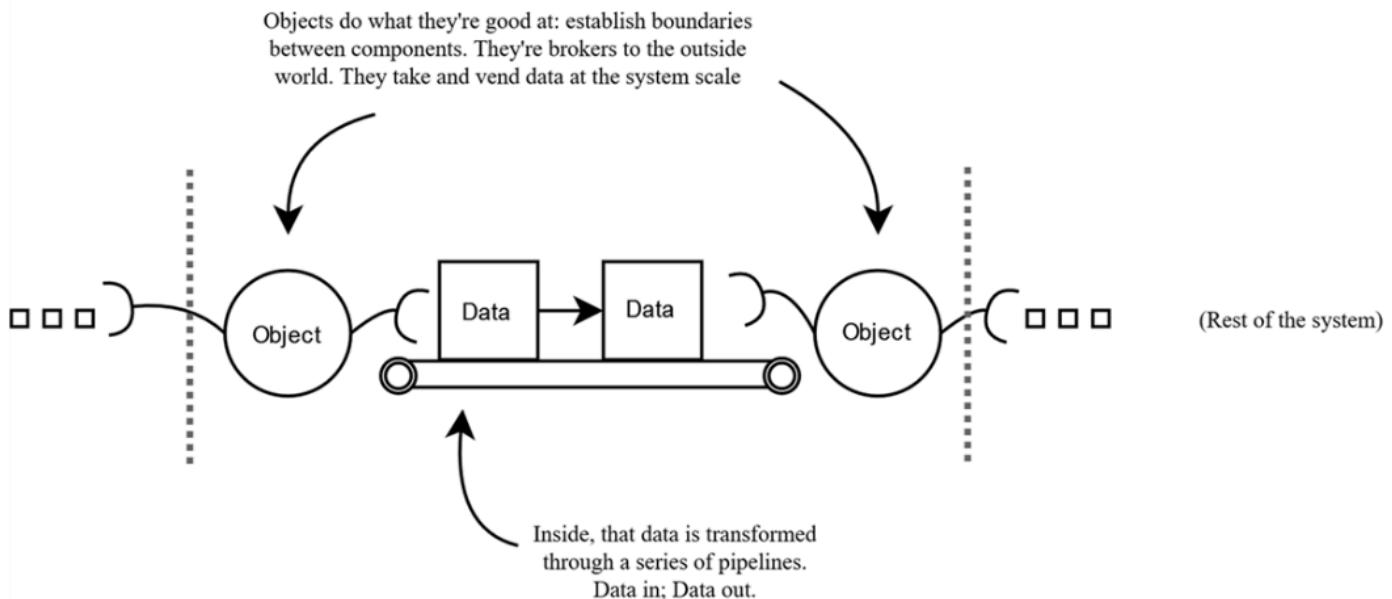


Figure 1.8 How data-oriented programs tend to be shaped

And that's pretty much it. That's the whole of data-oriented design. Step 1 is to figure out what you're really talking about -- what the data means or *is*. Step 2 is figuring out how to express that meaning in the code. Step 3 is letting those effects ripple outward through your program. The rest of this book is just an exploration of what that looks like in practice. It's a simple idea, but when it's applied thoughtfully, it can reshape how we program.

1.5 Which version of Java do I need?

The book primarily targets Java 21. That version gives us access to the latest tools like records, pattern matching, and sealed types. However, if you're not using these later JDK versions, fear not! You can still use every idea in this book even if you're on older versions of the JDK. The true "bedrock" version of the book is JDK 8. In my day job, I still regularly touch several codebases that are running this ancient JDK (so it holds a

special place in my heart). The ideas of data-oriented programming are not language or JDK specific.

Data-oriented programming is about the data, not the tools. So, even as we explore Java's new powerful data-oriented features, we'll always pause at each new addition to describe how we can accomplish the same thing even if you're running a version of the JDK that doesn't have those tools. We'll talk about which libraries to consider to close the feature gaps, along with where annotation processing fits into the story.

Sometimes, there won't be a great alternative at the tooling level, but, where the tools leave off, convention takes over. We'll make sure to cover patterns that can keep our code clean, communicative, and data oriented even on older JDKs. In all cases, the important part is the data. Better tools are just a bonus.

1.6 How does this book teach?

Rather than one big project that we expand on throughout the book, each chapter will mostly have its own self-contained examples (like this chapter, with its exploration of Scheduled Tasks). Most of these examples are pulled from things I've worked on in real life (though, of course, simplified to fit into a book).

The main reason we're using a bunch of small examples is that it gives me a chance to show you all of the ways in which even "simple" things can go wrong. If this book is one thing, it's a collection of my errors. My programming career has been steeped in mistakes, expensive design errors, and code so incomprehensible that it has led to full rewrites. I want to save you a bit of time (and personal shame) and show you how to avoid the problems that I've already encountered.

With that in mind, we'll often spend time purposefully going down the wrong path in each example to explore why it doesn't work. The goal is to build up an intuition for what it feels like when our modeling decisions are causing bad properties to emerge in the code. Often while programming, we encounter something that feels "off", but we can't really articulate *why* it feels off. By spending time milling around in that "off" state, we can learn to build up a grammar that can describe what it is about our individual design decisions that isn't working. It's from a thorough understanding of why something isn't working that correcting it becomes simple.

1.7 Wrapping up

So, that's data-oriented programming in a nut shell. Or, that's my definition of it, anyways. It's what it'll mean within the pages of this book. Outside of it, you'll find lots of other definitions (we're running out of catchy names to "orient" our programming around). Some of the other flavors focus on things like squeaking out every last drop of performance from the hardware, or standardizing certain programming patterns (by giving up static typing), but our version focuses on what the information in our domain *means*, how we can model that meaning in our programs using "data as data," and how doing so can transform our programming.

Next up, we're going to lay some ground work to explore exactly what we mean by "data" and how we model it with objects. We've been pretty loose with its definition throughout this chapter, so we'll need clear up what it is and how it's different from what we usually program with: identity and state

1.8 Summary

- Data Oriented programming is about programming with data "as data"
- Data is more than just a collection of values. It has an inherent meaning.
- Modeling "data as data" lets us focus on capturing that meaning in isolation from other concerns
- Before asking "what does it do?" data orientation starts a more bedrock question of "what is it?" We want to understand what these things in our domain are at a fundamental level

- Data Orientation is not a replacement for object orientation, functional programming, or any other paradigm. We view all of them as useful tools.
- The representations we choose for our data affects our programs as a whole.
- Good representations eliminate the potential for bugs by making it impossible to create invalid data
- Bad representations introduce problems which ripple outward through our codebase and force us to spend effort working around them
- We can replace reasoning about what vague variable assignments mean by representing that meaning with a concrete data type
- Focusing on the data inside of our objects, rather than just the interfaces, makes our objects as a whole more understandable
- When we do a good job of modeling the data, the rest of the code will feel like it's writing itself. We just have to follow where the data leads
- Data-Oriented programs tend to be built around functions that take data as input and return new data as output
- We'll use Java 21 throughout the book (though, you can still follow along with Java 8)

2 Data, Identity, and Values

This chapter covers

- Identity and change
- Values and Value Objects
- What “Data as Data” means
- Modeling data with Records

In the first chapter, we started to look at the effect that representation has on our programs as a whole. Each choice we make ripples outward (either spreading good effects or ill depending on our modeling choices). By really focusing on what something in our domain *is* - what it represents -- we were able to use that understanding to radically simplify our code as a whole. There's another question we have to ask when we're taking our first steps with data-oriented programming. It's a bit more fundamental, but holds equal powers to reshape how and *what* we program with. It's a little casual metaphysical question that goes like this: what does it mean to *be* something?

Programming with data as data forces us to tackle this question (at least superficially). One of the initial departures from the standard object-oriented toolkit is that when we're programming with data, we don't have the familiar container of the object to give the things we're modeling their "it-ness." So, the question becomes: what is the canonical *thing* that flows through our program? What do we, the people writing the code, structure our code around?

To answer this, we have to go on a bit of a quick philosophical tour of what it means to *be*. How we represent the identity of the things we're modeling shapes how our code comes together and how we reason about it. If you understand what identity is, and how we deal with it when modeling with data, you'll understand why data-oriented programs tend to look the way they do. It's not a stylistic choice, or because "that's how some guy with enough self-importance to write a book said it should be", the code and its patterns are emergent.

2.1 Identity is about continuity over time

Identity is how we track what all the continuously changing values that describe something *are*. It's what lets us track them as being related to one logical thing, even as they undergo (often massive) changes. For instance, the current me, a rapidly aging mid-30s guy, is about as far away as can be in every way from the “me” I was as a baby, but I'm still the same “me” regardless. The “me-ness” exists independently of whatever current set of attributes might be used to describe me (age, name, height, hair-remaining, level-of-interest-in-wood-working). I'll still be me – still have the same identity – regardless of how any of the individual attributes that might describe me change.

Identity is everywhere in our programs. Any time we ask a question like “what is the state of...” we're really asking about what that state is relative to some specific identity.

Identity involves a particular frame of reference. It could be some construct in our program ("what's the state of my object?"), or the program itself ("what's the state of the execution?"), the identity can also live "outside" of our program entirely ("what is the state of the payment?"). Even something as simple as a variable assignment sets up a surprisingly complex relationship between the identity of the variable we've created and the values that get stored inside of it over time.

Listing 2.1 A relationship between time and values

```
public static void main(String... args) {  
    double xPosition = 4.2;  
  
    xPosition++;  
  
    xPosition = xPosition * xPosition; #A  
}
```

#A The values assigned to xPosition keep changing, but the variable gives them a continuous identity

The variable, `xPosition`, has a current state, and that state is made up of some value, but *the variable is not the value it currently holds*. The variable is something different. It represents a special kind of "place" in the code where we can go to store and update values *about* `x` positions. The values we keep inside of this place will change over time, but the place itself will remain fixed. The place gives an identity to the different things we store in it. We can freely swap things in that place, but no matter what we put in there, it'll remain about this idea we've called `xPosition`.

One thing that's interesting to notice about this variable identity is that it doesn't really "exist" in a concrete way. Nothing about the variable itself as a "holder of things" is unique or tangible. It has no special tag or reference or "variable-ness" that we can introspect (without dropping down to the byte code) -- it's completely transparent to what's inside of it (which may have their own identity / notions of distinctness). Instead, we *impose* the identity on our variable as a useful mental model. We could technically use that variable to store information completely unrelated to `x` coordinates, but we don't (because that'd be silly). We use that particular name and that particular variable to give continuity to the related set of changes over time as they apply to this particular `x` position.

Java, being an object-oriented language, offers us another kind of identity. One that's not imposed by us, but one that's concrete and tangible at runtime. This identity comes from the object, and is the one that predominately shapes how we usually program and what our code looks like.

2.1.1 Object Identity

If we were going to model, say, a Person, objects let us give them a tangible "body" of sorts. And that body is what holds their "person-ness." Any attributes that describe our person, like their age, or name, or hair color are all associated with that specific body. But those attributes are not what makes the person *them*, they're just attributes of the body. We can freely change those attributes (even entirely!), without ever losing the core essence of the person. The object gives us a thread of continuity to our person even as the attributes which *describe them* change over time. The object gives our

person an *identity* that exists beyond their attributes. This identity of who our person *is* persists across the execution of our program. Object identity lets us point at the object and say, yes, *that person?* That's *our* person.

Listing 2.2 Each instance creates a unique object identity

```
class Person {  
    String name;  
    int age;  
    String hairColor;  
  
    // getters, setters, etc.  
}  
  
Person person = new Person("Chris", 36, "brown"); #A  
  
person.setAge(37); #B  
person.setHairColor("less-brown"); #B  
person.setAge(38); #B  
person.setHairColor("wait -- is that a grey..?"); #B  
  
person == person #C
```

#A Creating an instance sets up a relationship between the object and a set of changing attributes

#B That object is our person. We can carry it around our program, modifying it as we go.

#C It's still the same person down here even though we advanced it through time and changed its attributes

Object identity dominates how we usually organize and think about our programs in Java. It gives us a canonical *thing* which we can carry around as we write the code. They're what we do our programming "with." They're also the first thing we have to revisit when we start to program with data. Modeling data "as data" requires shifting how we think about what change in our programs means. This is because data doesn't have an identity like objects do. Because data isn't an object. Data is a *value*.

2.2 Values in the land of Identities

Values are special things. They transcend our objects, and programs, and computers entirely – values just *are*. The natural number 4 is a value. Its "four-ness" as a natural number exists regardless of how we express it (0100, IV, .4E01) or if we use it in our program at all. The number 4 just *is*. Further, because it just *is*, the number 4 cannot be changed – it doesn't even make sense to talk about it changing. Four is four! Values have no identity, because they have no notion of change. If I add some number to 4, it doesn't *replace* 4 (thus upsetting some kind of cosmic mathematical balance (which would be undesirable)), it *produces* a new number as a result, which itself is another unchangeable, transcendent value!

Listing 2.3 Delightful cosmic stability

```
4 + 1 #A
```

#A Thankfully, this doesn't delete the number four from the universe

Value classes are how we capture these idealized values within the confines of Java. `Integer`, `Double`, and `String` are all great examples of value classes built into the language. They still produce objects, of course, and thus technically have an object identity

(because they get assigned one by nature of them being objects in Java (though this will soon be changing. Checkout JEP 401)), but that object carries no *semantically meaningful* identity in the sense that we've been exploring. Their object identities are an unimportant implementation detail. In fact, things can behave unexpectedly if you try to care about the *identity* part of the object part rather than the *value* part.

Listing 2.4 Depending on object identities for value objects can lead to strange results

```
Integer x = Integer.valueOf(128);      #A
Integer y = Integer.valueOf(128);      #A
System.out.println(x == y); // FALSE #B
```

```
Integer x = Integer.valueOf(127);      #C
Integer y = Integer.valueOf(127);      #C
System.out.println(x == y); // TRUE #D
```

#A We're creating different objects, which means unique object identities, right?

#B So far so good. This checks out.

#C But, hang on...

#D What the -- Now these "different" objects have the same identity!? Java secretly caches and reuses certain objects to optimize performance. Rely on object identities with caution!

Object identity is something you should forget exists for value objects. It's there as an artifact of the Java language, but it's irrelevant to the modeling of *values*. That identity should be free to be optimized away, cached, reused, or otherwise mangled in various ways. It's irrelevant to us, because value objects don't "have" an identity as far as we're concerned. There is no change to track, so no identity needed. Once they're created, they *are* that value. Forever. Until the end of time.

Since values have no identity, what they "are" is determined entirely by their *state*. However, note that "state" in this case does not refer to *mutable* state. Value objects have exactly one state -- the one we give them during creation. If two values objects have the same state, then they are the same value, regardless of whatever kind of identity might be "holding" them.

Listing 2.5 A value's "state" determines what a value "is"

```
Integer a = Integer.valueOf(3042);
Integer b = Integer.valueOf(3042);

assert a.equals(b); // true      #A
assert a == b;     // false     #A
```

#A a and b are the same value, even though they're assigned to different objects, and different variables, and have unique object identities.

In listing 2.5, everything about the variables `a` and `b` says that they're different objects. They have different object identities, they're assigned to different variables (thus, potentially have a unique imposed identity), but, none of that matters, because their states are equivalent to each other. Value objects are special *because* their container doesn't matter. We can substitute `a` for `b`, or replace every instance of either variable with a brand-new value object with *no observable change in our programs meaning or behavior*. Value classes, when designed correctly, are transparent to what's holding them.

Listing 2.6 Many objects; all of them the same

```
Integer a = Integer.valueOf(1234);
Integer b = Integer.valueOf(1234);
Integer c = Integer.valueOf(1234);

assert a.equals(b)                                #A
  && a.equals(c)                                #A
  && Integer.valueOf(1234).equals(a)            #A
  && Integer.valueOf(1234).equals(b)            #A
  && a.equals(Integer.valueOf(1234))           #A
  && Integer.valueOf(1234).equals(Integer.valueOf(1234)) #A
  && a.equals(b) && a.equals(c)                #A
```

#A All of these are equivalent because they represent the same value

2.2.1 Building values out of values

Value classes aren't restricted to simple scalar values like numbers. They can be arbitrarily complex. We can build new value classes on top of existing ones.

Listing 2.7 Building new Value objects from existing value objects

```
class Vector {          #A
    Double x;          #A
    Double y;          #A

    public Vector(Double x, Double y) {
        this.x = x;
        this.y = y;
    }
    public Double x() { return this.x; }      #B
    public Double y() { return this.y; }      #B
    public boolean equals(Object other) {...}
    public int hashCode() {...}

} #C
```

#A We're creating a new value object that's built from other value objects

#B Note that while we've got accessors, we don't have any setters! Values do not -- cannot! -- change.

#C There's technically one more thing we'd need to do to this class to make it a true Value Object, but we'll get to that below

Values classes remain values as they grow in complexity as long as they honor the rules we've established so far: values class must be immutable (values just *are*. They do not change), and their *equality*, must always be based on their state. A value is described entirely by the attributes which make it up. This is easy to see with individual scalar values like 4, but is something we have to enforce in Java for compound values like Vector.

Listing 2.8 Many ways of saying the same thing Part II

```
Vector a = new Vector(2.0, 3.0);
Vector b = new Vector(2.0, 3.0);
Vector c = new Vector(2.0, 3.0);

assert a.equals(b)                      #A
&& b.equals(a)                        #A
&& a.equals(c)                        #A
&& new Vector(2.0, 3.0).equals(a)      #A
&& new Vector(2.0, 3.0).equals(b)      #A
&& a.equals(new Vector(2.0, 3.0))     #A
&& new Vector(2.0, 3.0).equals(new Vector(2.0, 3.0)); #A
```

#A Even as the complexity of our values grows, their equality semantics must remain the same

As long as we honor the semantics, the value objects we create from this Vector class will be as value-like as any other “simple” scalar value class in Java (like Integer and Double) despite being more complex. We can reason about it in the exact same way. We’ll be able “see” those value objects as *being* the values that they represent (or as close as we can get to them within the confines of our language). (The power that comes from seeing “through” objects into the concrete values they denote is something we’ll explore in detail in the next chapter)

There’s also no upper bound to this. Value objects can be as complex as we need them to be. There’s a tendency amongst Java developers to mentally bucket value objects as being restricted to “simple” things that consist of only a handful of attributes (Points, Vectors, Rectangles, etc.), and that there’s a loose cut-over point beyond that where traditional objects need to take over. This is usually born from feelings of pressure around “information hiding” or “managing complexity” or mumblings about the Law of Demeter. However, the “best practices” for the modeling of objects doesn’t apply to the modeling of values. We have to learn to quiet that deeply ingrained voice which wants to judge everything through the identity object focused lens. *Value classes* are different from all other classes. We have to judge them with a different set of eyes.

This no-limit-on-how-big-a-value-can-be idea similarly applies to *collections* of values. Lists, sets, and Maps can all be as value-like as the number 4. However, seeing collections this way can take some squinting at first. By and large, we interact with the Java collections as identities. A lot of our programming patterns are built around this identity. Collections are carried around as we program and built up over time as we gather more information.

Listing 2.9 Carrying around a mutable collection as we perform work

```
List<Customer> customers = new ArrayList<>(); #A

QueryResponse response = database.query(request);
for (Result result : response.items()) {
    customers.add(Customer.from(result)); #B
}

while (response.nextToken() != null) {
    response = database.query(request, response.nextToken());
    for (Result result : response.items()) {
        customers.add(Customer.from(result)); #C
    }
}
return customers; #D

#A Creating the collection gives us a useful identity we can carry around as we do work
#B Which we do here
#C And then over-and-over again down here
#D Before finally returning it here
```

However, despite how we might commonly use these things, identity is a modeling choice we get to make while designing our software. We get to decide what identity we'll imbue into our collections -- including giving them none at all. The Java collections are flexible enough to allow us to represent them as both *identity objects* or *value objects*. It's up to us to choose what we want our collections to *mean*. Are they objects which give an identity to some changing set of values over time, or are they immutable and unchanging values?

2.2.2 Treating Java collections as Values

The main tool for turning Java objects into values is the suite of “unmodifiable” wrappers in the Collections package. These take a mutable Java collection (an object with an identity) and turn it into a value by disabling all of the APIs which would allow modifications.

Listing 2.10 Turning mutable collections into values

```
List<String> letters = new ArrayList<>(); #A
letters.add("A"); #A
letters.add("B"); #A
letters.add("..."); #A
letters.add("Z"); #A

List<String> immutableLetters = Collections.unmodifiableList(letters); #B

immutableLetters.add("1"); // ERROR #C
```

#A The default collection constructors create mutable identities
#B But we can convert them into values when we're ready via the unmodifables
#C Any subsequent attempts at modification will kick us out with an error

As of Java 10, we can create these Unmodifiable collections directly thanks to the `.of()` methods available on each of the major collection interfaces. These simplify the process of creating collections as values.

Listing 2.11 Different constructors yield different object semantics

```
Set<Integer> a = Set.of(1, 2, 3, 4);          #A
List<Integer> b = List.of(1, 2, 3, 4);        #A
Map<Integer, String> c = Map.of(1, "One", 2, "Two"); #A
```

#A All of these create collections as values

The Unmodifiable wrappers handle the “does not change” part of being a value. For the equality side of behaving like a value, the collections conveniently already do the right thing. Their equality is based on the state of what’s inside of them. So, if the collection is made up of values, and the collection itself cannot change (because we’ve used an unmodifiable flavor), then that collection represents a value object.

Collections that follow the value “rules” are as value-like as any other value. Because they can’t change, they’re safe to use in places where collections as identity objects wouldn’t be safe to (or would potentially lead to confusion).

Listing 2.12 Perfect safe, as long as we’re dealing with values

```
Map<List<String>, String> map = new HashMap<>();

map.put(List.of("Bob", "Joe"), "Bob and Joe");    #A
map.get(List.of("Bob", "Joe"));                    #B
// [out] "Bob and Joe"

List<String> mutable = new ArrayList<>();          #C
map.put(mutable, "TBD");
mutable.add("Bob");
map.get(mutable);                                #D
// [out] NULL
```

#A Perfectly safe, because it's a value

#B We can query it back out using a totally different value object, because their equality is the same (i.e. they're the same)

#C However, with an identity-based setup...

#D You'll probably never be able to find this object in the map again.

The flexibility of the Java collections is a double-edged sword. The same data type can represent both an identity object as well as a value object. Which one you get is entirely determined by which constructor we call while creating the collection. This means that it’s remarkably easy to end up with different semantics than intended just by calling a slightly different form. Worse still, is that the two worlds aren’t exactly clear cut. There are a few gotcha constructors, like `Arrays.asList(...)` that initially appear to produce values (you cannot resize the array), but still retain their internal mutability (items in the list can be reassigned).

Table 2.1 A small (non-exhaustive) comparison of the semantics of List's different constructors

Produces an Identity	Produces a Value
<code>New ArrayList();</code>	<code>List.of(...)</code>
<code>Stream.of(...).collect(Collectors.toList());</code>	<code>Stream.of(...).toList();</code>
<code>Arrays.asList(...);</code>	<code>Collections.unmodifiable(list);</code>

This problem of semantics changing during creation isn’t unique to collections. Java’s mutable-by-default behavior means that creating anything in Java runs the risk of

inadvertently creating something with an identity. Some are less obvious than others. Learning to see them will help keep your values as values.

EXPLORING COLLECTION OUTSIDE OF JAVA'S STANDARD LIBRARY

Java doesn't currently expose its Unmodifiable suite of data types publicly (*cough* though it'd be great if it did *cough* (nudge. nudge)). So, we might initially appear to be stuck with the task of remembering to call this or that particular constructor in order to give ourselves the appropriate object semantics. However, not all is grim! Later in the chapter, we'll explore some of the tools Java gives us for working around the default mutability of the standard collections.

The other option worth exploring is checking out some of the third-party collection libraries available in the Java ecosystem. Guava is an extremely popular "drop-in" replacement. They implement the standard Java collection interfaces, but with the big modeling benefit of exposing their immutable implementations for us to use in our modeling. Another option, and my personal favorite, is a library called Vavr. It departs from the Java collection interfaces, and thus can involve a lot of interop work, drawing new boundaries in the app, and retraining developers on the team, but if those cons don't push you away, the reward is a suite of immutable collection types with an extremely unique take on what APIs should look like when modeling values.

2.2.3 Things to watch out for when creating values

Creating values requires some special care in Java. It's very easy during object creation to inadvertently shift from "creating a value" to "creating an identity that holds values over time". The problem stems from what we assign our values *to*. No matter how concrete, immutable, and theoretically pure our value class is on its own, all of its value-ness gets destroyed as soon as it's assigned to a mutable variable. The number 4 just *is*, but the number 4 assigned to a mutable variable *is not*. The act of assignment creates a *relationship* between that variable and all possible values that could ever be assigned.

Listing 2.13 This does not create a value

```
Integer xPosition = Integer.valueOf(4); #A
```

#A The variable can be reassigned. This doesn't create a value, it creates a time-varying relationship between a set of values and the identity we call xPosition

The variable `xPosition` is *assigned* a value, but *it is not the value*. The act of assigning it subtly changed its semantics. The assignment created an identity under-which some *set* of values can freely be reassigned throughout the execution of the program. All of the good stuff that makes values so useful (their not changing) gets lost when we assign it to a mutable variable. The integer itself is still a value object, but where it's assigned is not. The value has been turned into the variable's mutable state.

This same thing happens with the instance-level assignments inside of our classes. Even if we intend to model something as a value, any time we assign values to mutable instance variables, we risk destroying its core value-ness and creating a bunch of individual identities tracking sets of values over time.

With this in mind, we can turn a critical eye towards the value objects we've modeled so far in this chapter. We left something important out of the `Vector` class in listing 2.7, and, in doing so, undermined our intention to create a new value. The problem is that our "value" is built on top of a foundation of mutable variables. Even though we're building the vector object out of other value objects, the assignments we make inside of the class drag the values back down into the world of state.

Listing 2.14 Where we left off

```
class Vector {  
    Double x; #A  
    Double y; #A  
}
```

#A Even if we don't make setters, nothing about the code explicitly says that this class is a Value Object

We worked around this short coming by not making any setters, but the problem remains that when we look at this code, nothing about it communicates to us as readers what its semantics are -- what it's supposed to *be*. Right now, those semantics only live in our heads. We, the current developer, "know" it's supposed to be a value, but we haven't put that knowledge into the code. This class looks like any other class we might find in the codebase. There's nothing that makes it explicit that *this class represents a value class*.

Clear semantics matter for the long-term health of the codebase. Time will pass, team members will come and go, and corporate political turmoil will completely upend ownership. Throughout all of that, the code will remain. It has to be able to describe itself. If we don't make it clear what we mean, then all of our intentions around what the code represents risks getting lost as the codebase changes hands.

Listing 2.15 Does this new method fit with the original intention of the class?

```
class Vector {  
    Double x; #A  
    Double y; #A  
  
    public void scale(double amount) { #A  
        this.x = this.x * amount; #A  
        this.y = this.y * amount; #A  
    } #A  
}
```

#A A perfectly reasonable addition if we're designing this class around a single object identity, but a completely invalid addition if we're modeling a Value Object.

We have to communicate what we want our code to mean to the compiler and the people who use our code. We have to express how we want to control change. To get our objects to retain their value properties, we have to make our objects, along with the variables we assign them to, unmodifiable. This is the role of the `final` keyword.

From our data-oriented perspective, `final` is a tool for communicating semantics. It lets us control what assignment *means*, or, more specifically, what assignment *is*.

Listing 2.16 Marking as final changes what assignment means

```
final Integer xPosition = Integer.valueOf(4); #A  
#A making xPosition final
```

By making the variable final, we've prevented the assignment from creating an identity that tracks some set of values over time. This variable *is* the value that's assigned to it. `xPosition` is now just another name for the integer 4. It will never change -- it cannot change. The final keyword means that it no longer makes sense to ask for the "current state" of `xPosition` anymore than it makes sense to ask for the "current state" of the number 4 itself -- it doesn't have a state -- it's a value!

Adding final to our classes has the same effect.

Listing 2.17 Adding final everywhere

```
final class Vector {      #A  
    final double x;      #B  
    final double y;      #B  
}  
  
final Vector vector = new Vector(2, 3);
```

#A We mark the class as final, too. Inheritance of values can be done (with a lot of caution), but avoiding it cuts off a lot of potential problems
#B Marking the instance variables as final

Now this is a true value!

OK, ALL THAT SAID...

When you're getting your feet wet with data-oriented programming and working with values, final is a very useful tool. In addition to keeping us honest as we program it also serves to communicate our intention about the semantics of the code to our readers. However (with a heavy dose of "just my opinion"), I think they're safe to drop once immutability is cemented into your local programming culture.

Immutability can be managed by convention when the team is all in alignment. In my own coding, it's pretty rare for me to punch out the final keyword anywhere unless the extra explicitness feels warranted. Local team culture runs with an "assumed final unless otherwise stated" kind of disposition. (Plus, upcoming versions of Java will soon give us new tools for creating value classes that manage the finals implicitly on our behalf.)

In that spirit, I'll omit them everywhere in the book in order to help keep the examples terse enough to fit on a page. Convention wise, though, when we're modeling values and assigning them to variables, we assume finality unless otherwise stated.

2.2.4 Mutability converts values back into identities

Values must be made up of values. As soon as we allow any non-values into the mix, we risk undoing all of our hard work. A single mutable reference is all it takes to pull us

out of the world of values and back down into the world of identities and change.

Listing 2.18 Even a single mutable addition prevents us from modeling values

```
final class Person {  
    final String name;  
    final Date birthday; #A  
}  
  
Person person = new Person("Bob", new Date());  
System.out.println(person);  
// [out] Person[name=Bob, birthday=Sun Aug 11 22:42:57 PDT 2024] #B  
  
person.birthday().setTime(new Date().getTime() + 123456789); #C  
System.out.println(person);  
// [out] Person[name=Bob, birthday=Tue Aug 13 09:00:34 PDT 2024] #D
```

#A Even innocent additions can accidentally convert value objects back into identities. Date is a mutable type!

#B We can no longer rely on our data type as a value. What it says now might not be the same later.

#C Anyone can reach inside of our object and jiggle its mutable references

#D Which means we're back in the world of mutable state. Each time we look at our "value" we might find something different.

Mutability poisons value objects. The two ideas cannot coexist. Mutable references on a value class reintroduces the notion of identity to something which shouldn't have one. This unexpected identity undermines all of the stability that values give – because the objects aren't modeling values anymore. They've been converted into identities. Trying to use them as values can lead to very unexpected program behaviors.

Listing 2.19 That's not how sets are supposed to...

```
Person person = new Person("Bob", new Date()); #A  
Set<Person> people = new HashSet<>(); #B  
people.add(person); #B  
people.add(person); #B  
System.out.println(people); #C  
// [out] [Person[name=Bob, birthday=Sat Aug 24 13:14:20 PDT 2024]] #C  
  
person.birthday().setTime(new Date().getTime() + 123456789); #D  
people.add(person);  
System.out.println(people); #E  
// [out]  
// [Person[name=Bob, birthday=Sun Aug 25 23:31:57 PDT 2024], #E  
// Person[name=Bob, birthday=Sun Aug 25 23:31:57 PDT 2024]] #E
```

#A Accidentally creating a value with a mutable "backdoor"

#B It appears to behave value-like at first. We can put it into a set over and over

#C And, of course, only get one instance of the value in the set

#D But then some unexpected mutation occurs and...

#E Things go off the rails. The basic contract of a sets no longer holds. We've bamboozled it into keeping multiple copies of the same "value"!

This poison is infectious. It converts everything it touches back into an identity no matter how many steps removed it might be. If you add a "value object" with a mutable reference to a list – even if you make that list Unmodifiable – the whole thing gets converted back to an identity as a result. You're only as value like as the least value-like thing you touch.

Listing 2.20 Poison runs deep

```
List<List<Person>> people = List.of(List.of(person));          #A
int original = people.hashCode();
people.get(0).get(0).birthday().setTime(12345678);           #B
System.out.println(original == people.hashCode()); // FALSE    #C
```

#A We used the right value-based constructor

#B But that doesn't matter, because inside of that "unmodifiable" list, just a few levels down, sits a mutable reference

#C If you change anything, you've changed everything

2.3 Data is what we get when we give values meaning

When we talk about modeling data "as data," what we're really talking about is modeling data as *value classes*. Specifically, data *is* a value. It follows all of the same rules (unchanging and immutable) and has all of the same expectations (it has no object identity, what it "*is*" is determined entirely by its state equality).

However, data is more than "just" a value. Data is special because it has a meaning.

The number 4 as a value just is. It's a quantity, a relative worth, a magnitude. On its own, it isn't data. It becomes data when we make that value about something. Hourly unique users, total sales, temperatures, days elapsed, etc. etc. etc. These things are all data.

Listing 2.21 Making the number 4 "about" something

```
Integer temperature = 4;      #A
```

#A Notice how much assigning it to something called "temperature" changes what that 4 means

What's crazy is how much the meaning of the value 4 changes as soon as we associate it a name like "temperature." That 4 is now imbued with context. It has picked up a *semantics*. It's now "about" something. It's now *data*.

With this new context comes an immediate rush of new questions. What kind of temperature are we talking about here? Does "4" mean something really hot or really cold? Would 0 represent absolute 0 or is it just an arbitrary point we can go below? What does this thing we've called temperature mean?

A huge part of data-oriented programming is noticing just how different that 4 feels once it was "about" temperature. We're suddenly aware of all the things our representation doesn't express. We're forced to ask all of these questions because there are now two distinct semantic levels going on in our program. The first is about the semantics of the data itself – "what kind of temperature are we talking about here? Fahrenheit? Kelvin?". The second is the semantics of the model of the data in our code ("does this code capture the temperature we're modelling?"). These two interplay in complex ways.

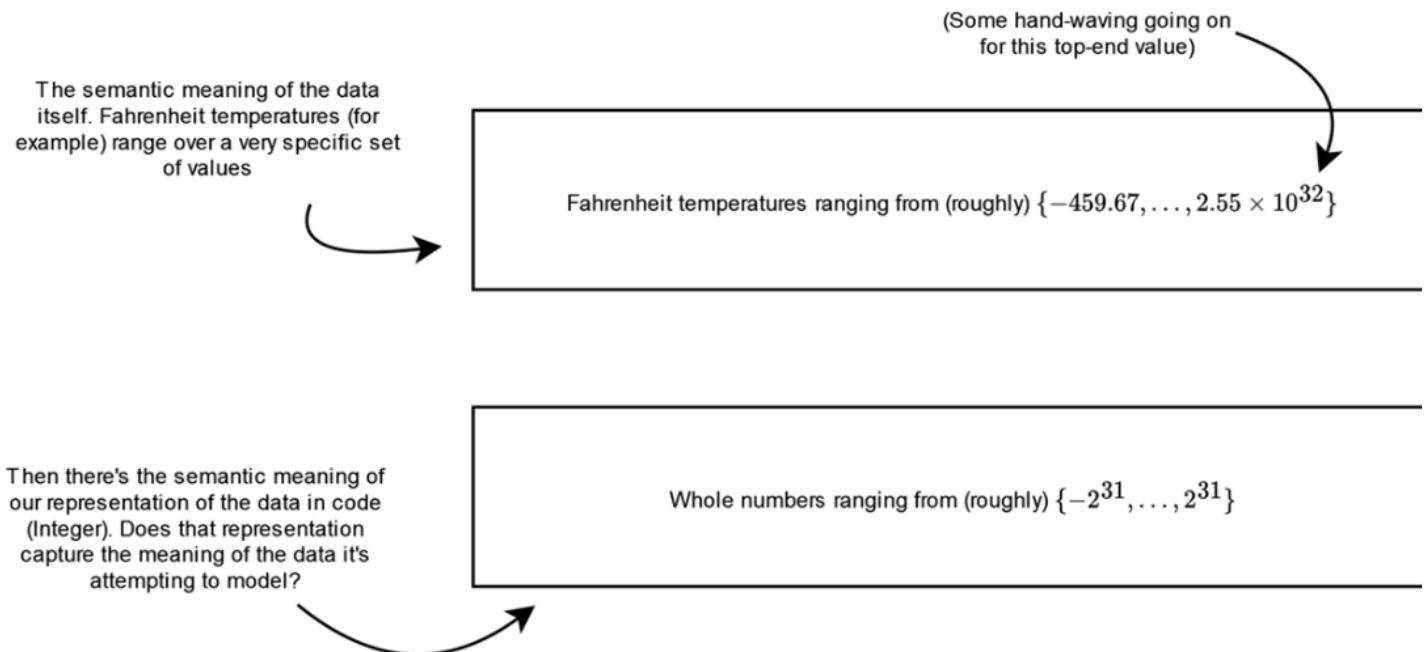


Figure 2.1 Comparing semantic levels

Data's meaning comes from outside of the code. It lives in a particular domain and will be filled with all kinds of constraints and quirks and unique aspects which make that data *that data*. The engineering work on our side is in *understanding* it. We have to talk to people (ugh...), read, and ask questions until we get to the bottom of it. In the case of this temperature thing we're modeling, we have to understand what it is, what unit it should be, how it's used, and what fidelity we need. Only then can we move onto to the next part: getting everything we just learned *out* of our heads and *into* the code.

This is where the soul of data orientation lives: representation. The types we use to represent the data in our domain have their own meaning and semantics. Data-Orientated programming is predominately being worried about what the types we choose communicate to our readers (and compiler!). Code is the medium in which we have to express ourselves. Little changes in representation have massive impacts on what our code conveys. For our temperature data, even we don't even know what unit it should be yet, we can still manipulate our reader's understanding of what it *could be* just by picking different data types.

```
Integer temperature; #A
Double temperature; #A
```

#A How different are these even though we know nothing about them yet?

The kinds of programs we could ultimately express are wrapped up in those two data types. One conveys discrete values. Coarse and low fidelity. My mind naturally drifts towards things for human consumption. For instance, oven temperatures in a recipe, an AC control panel, or daily weather reports. I want to know if it's 71 or 72 degrees. I wouldn't know what to do with 71.3 degrees. The other, double, is continuous, and conveys (to me) that fidelity of the measurement matters. Whatever it is we're modeling, the difference between 70.1 and 70.2 is significant. We'll care about it because the code says we'll care about it. This is the power that data modeling has.

2.3.1 What can data be?

A lot of data takes on the very familiar shape of being *about* something in the world – sales totals, quarterly reports, daily weather, active users, etc. Stuff we might just sum up as: *information*. Generally, we find this type of data in files that we process, network requests that we receive, or the usual goings on of a business that we retrieve from a database.

Listing 2.22 data that looks like “data”

```
class DailyElectricityUsage {  
    LocalDate date;  
    Temperature high;  
    Temperature low;  
    Double kWh;  
}  
  
class Sale {  
    String productName;  
    LocalDate soldOn;  
    int quantity;  
    BigDecimal totalPrice;  
}
```

This flavor of data is probably what pops in most of our minds when we think of the word “data.” It’s generally something that’s made up of *records* -- facts or events about the something that happened (items purchased, user logins, kilowatt hours used, etc).

However, data isn’t restricted to the world of records and information processing. Data can be used to model abstract ideas like decisions and actions. We saw this in chapter 1 where we modeled the actions our program could make in response to certain events as plain data. We created data that was about the logic of the program itself.

Listing 2.23 data about an action the system can take

```
class ReattemptLater {  
    LocalDateTime scheduledAt;  
};  
class RetryImmediately {  
    LocalDateTime scheduledAt;  
    int attempts;  
};
```

Sometimes even just a name on its own can be data. We also saw this in the previous chapter. The `Abandon` data type wasn’t built on top of any other value classes. In fact, it wasn’t “made up of” anything at all. The entire “data” was just the name we gave to the type. It represented an *idea* as a value.

```
class Abandon {  
    #A  
};
```

#A Abandon isn’t made up of anything else. Its name, “Abandon,” is the data.

There’s no limit on what data can be or what we can model with it – because data is just a value to which we attach some meaning. Data can even be used to model state

itself. It can be “about” an identity. Even though data is built on top of unchanging immutable values, we can use that data to *model* change. When we do this, we unlock something extremely powerful and expressive. It opens us up to modeling the core entities in our domain, with all of their rich life-cycles and change over time, as plain, unchanging, immutable data. To do that, we just have to revisit how we think about what identity means.

2.4 Identity in the World of Data

Identity is what gives continuity to a succession of changing values over time. We usually model this identity using constructs provided to us by our language, which, in Java, usually means object identity.

Listing 2.24 Object identity

```
Person person = new Person("Bob", 32);
person.setAge(33);                                #A
person.setName("Bobbie");                          #A
person.setName("Robert");                         #A
```

#A Updates are done to the object. It gives an identity to all of those changing values

However, identity isn’t tied to a specific language construct. Objects give us an identity, and we’ll even say informally that the object “is” the thing we’re modeling, but the object itself isn’t really the “true” identity of the thing we’re modeling (in the same way that the type Integer isn’t the same thing as the value 4. One is a useful approximation of the other). It’s just a model after all. An approximation we make with code. The identity exists regardless of how we model it, or which feature (or language) we use to model it.

If we simulated tossing a ball in the air, but then paused that simulation midway through, dumped that state to a file, sent it across the network, and restarted it on a completely different computer, the ball would continue along on its journey with no idea that anything had changed. It didn’t become a different ball when it changed machines, objects, address spaces, or CPU architectures, it’s still the same “ball” from an identity perspective even though what’s simulating it is different. We’re simulating *that* ball.

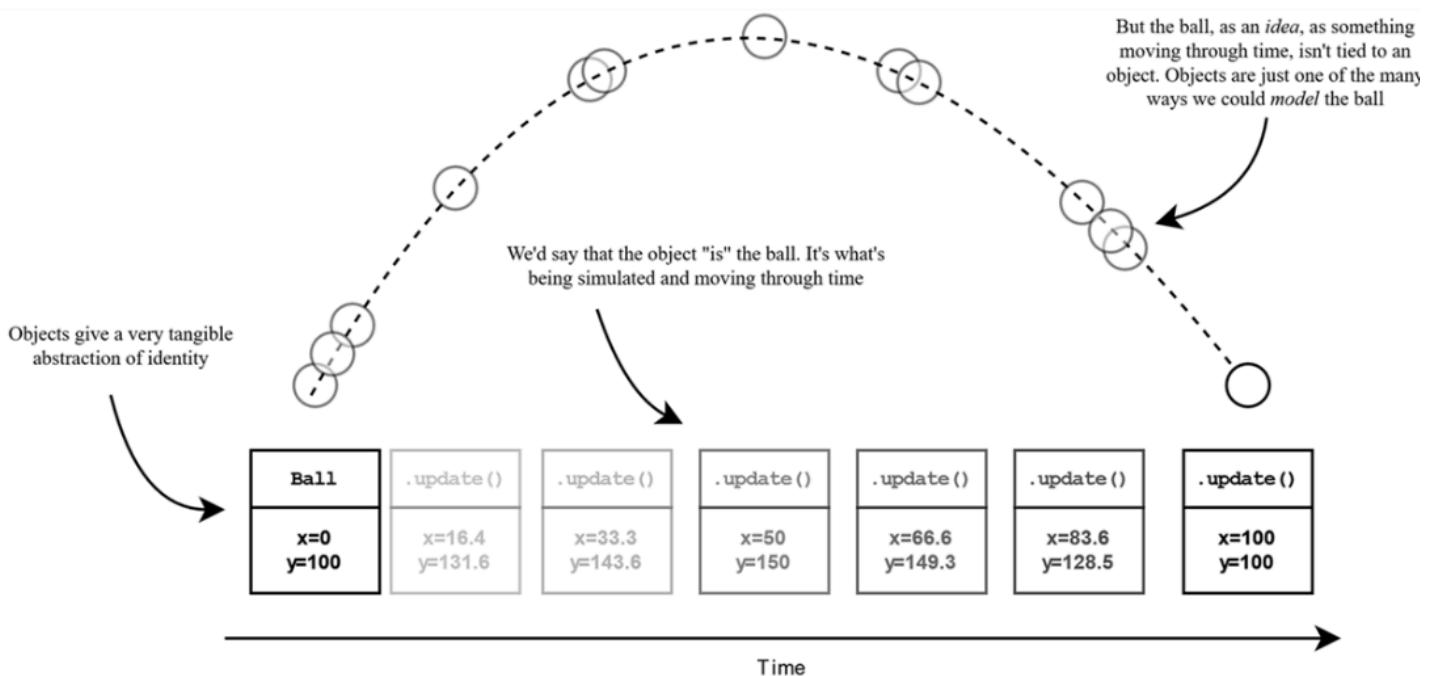


Figure 2.2 Simulating a ball being tossed. The object is a useful abstraction, but the idea of a ball exists on its own

The question here is how much we really need the object, or any particular language construct (like a mutable variable), to represent the identity of this ball that we're trying to simulate. Identity is about giving continuity to a *succession of values* over time. Inside of the object, there are *values* flowing through it. We use the object to give those values their continuity, but what would happen if we just looked at all of the values on their own? Is it only because they're updated in place in our object that they become "about" this identity? Is continuity between these values entirely dependent on mutation?

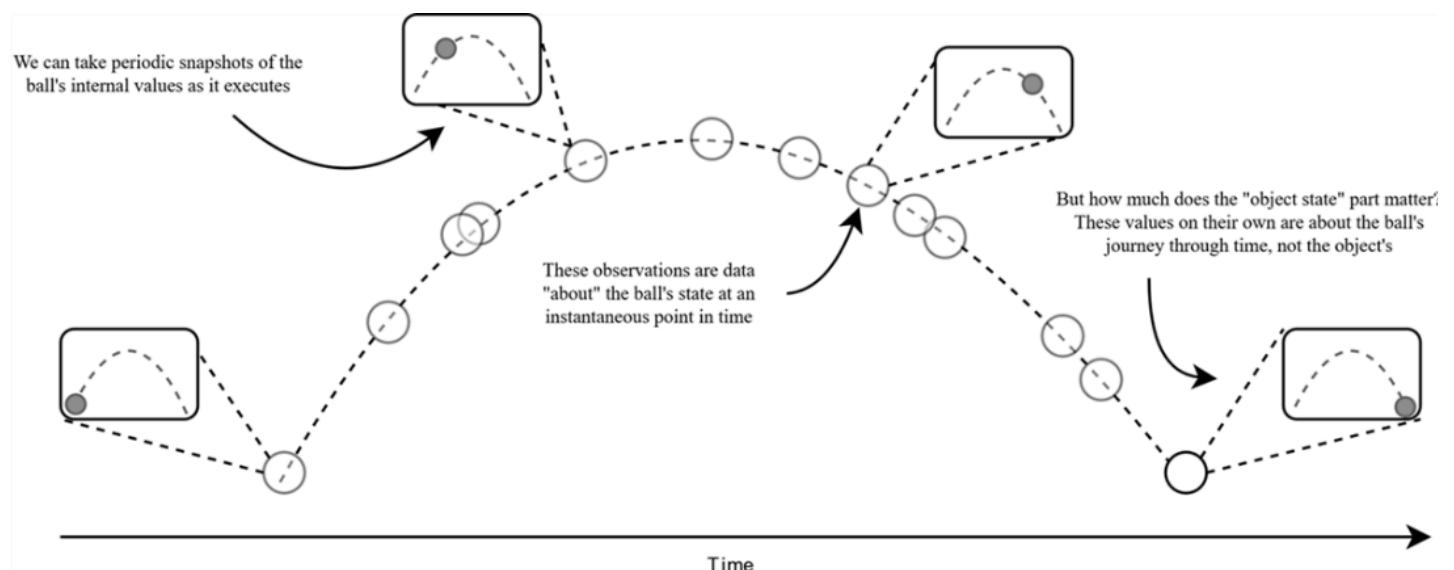


Figure 2.3 Looking at the succession of values inside of an object

If we take all of the values that get assigned inside of the ball instance and lay them out on a timeline, you could imagine that we'd end up with something that looks kind of like a film strip. These "frames" each describe a state that the ball entered (or will

enter). Further, we could imagine playing these frames back, one after the other, to “simulate” this ball being thrown through the air.

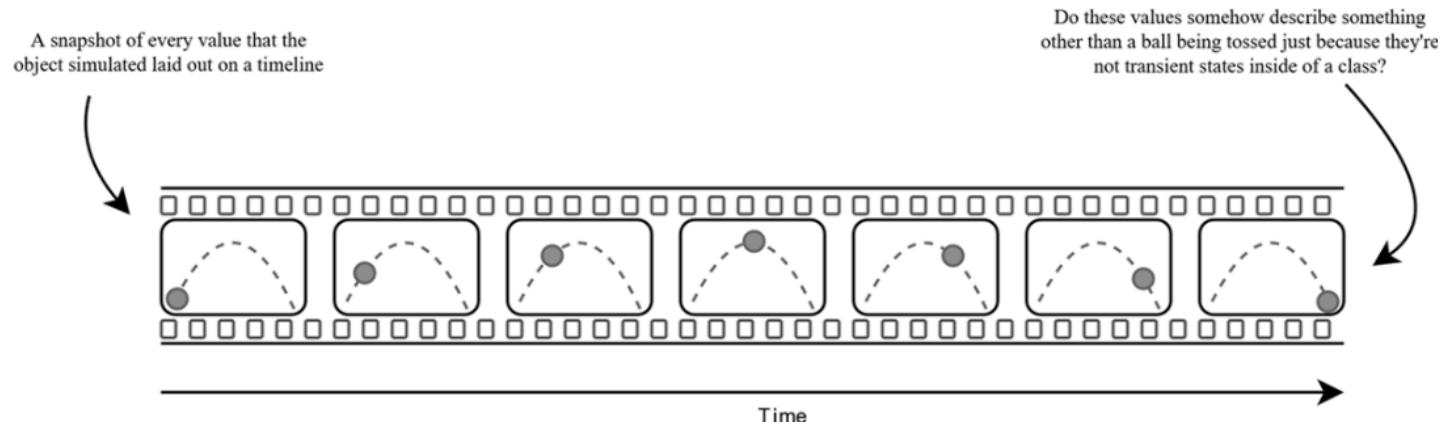


Figure 2.4 A succession of values laid out on a timeline

From the point of view of the thing being simulated, a ball in this case, the result of that playback would be indistinguishable from the continuous stateful version from which we grabbed these values. Similarly, to the people observing the outward effects of those changes (the ball moving to a new location), there'd again be no difference. This is, after all, how movies work.

We don't need to modify things in place to represent change any more than a film does. State is *emergent* from the successive transformation of data. To us, the programmers working with this data, the identity is simply *imposed*. These values in the code are about the same thing *because we say they are*. It doesn't need to be any more complicated than that. The “true” identity, what these things we’re modeling “are,” is realized outside of our program – on a movie screen (to keep with the film stock example), or in the database when we persist the changes, or when shipped across a wire to some authoritative system of record, updated on a UI, or otherwise just side-effected out to the real world in some way. Identity is that thing we call the continuity that comes from a succession of values. We’re free to model it however we like.

2.4.1 Identity shapes how we program

Using values to model identity and change over time requires shifting some of “how” we program. Specifically, what we program “with.” Values cannot change. They have no identity of their own that we can carry around as we make modifications. Once we create them, they’re that value forever. Because of this, data-oriented programs tend to take on a different shape from their purely object-oriented counter parts. This isn’t because of some guidance from on-high, or “how I say it should be done,” the programming style emerges naturally from the constraints that programming with values imposes.

To explore this a bit, let’s go back to the Person class we’ve been toying with throughout the chapter. We’re going to explore how code structure evolves when we model change as a series of values rather than implicit in-place updates. The “change” we’ll be modeling in this case will be simple: giving our person a birthday.

We’ll start with the object-oriented flavor.

Listing 2.25 having a birthday with objects

```
class Person {  
    String name;  
    Integer age;  
  
    public void haveBirthday() { #A  
        this.age++; #A  
    } #A  
}  
  
// ...  
  
Person person = new Person("Bob", 32);  
person.haveBirthday(); #B  
person.haveBirthday(); #B  
person.haveBirthday(); #B  
System.out.println(person);  
// [out] Person(name="Bob", age=35); #C
```

#A This method takes nothing and returns nothing. It mutates the internal age state in place
#B We can repeatedly call the same method on the same object. Each time it updates our person to a new state
#C The “person” (our object) has changed. Their original age is lost to time – they’re now 35.

Object identity rules the land with the usual approach. The “shape” of the code – its physical layout on the page, is built around doing things “to” objects. Once we’ve got an object in our hands, we carry it with us around the code. It’s the object that does the changing. The methods on that object are how we do it.

A very different programming style emerges when we model the Person entity as data.

Listing 2.26 Modeling person as data

```
final class Person { #A  
    final String name; #A  
    final Integer age; #A  
    // equals, hashCode, etc.  
  
    public void haveBirthday() { #B  
        this.age++; #B  
    } #B  
}
```

#A Modeling our person as an immutable data type (we’ve added the explicit finals to make it clear that we’re talking about data)
#B Because it’s now data, we can’t keep the birthday method as originally implemented. There is no mutable state left for it to modify!

What forces us to adopt a different programming approach is that *data doesn’t change*. As soon as we create an instance of our Person, it’s “stuck” as that particular value forever. We can’t do anything to this object by calling methods. We can’t really “do” anything at all, so... how do we represent change with things that can’t change?

Listing 2.27 Uh... now what?

```
final Person person = new Person("Bob", 32); #A
// uhh...
```

#A Once values are created, they cannot change, so... where do we go from here?

We represent change by creating more data! We can model that identity through a *succession of related values*. It's through all of these little "frames" that the change as a whole emerges. We can sketch that change out manually by creating an individual piece of data that each represents the person growing older and older.

Listing 2.28 manually creating "frames" that represent our person over time

```
final Person person      = new Person("Bob", 32); #A
final Person older       = new Person("Bob", 33); #A
final Person evenOlder   = new Person("Bob", 34); #A
final Person cruelTime  = new Person("Bob", 35); #A
```

#A Nothing is mutated, but we can still see the steady progression of change. They're all clearly "about" the same identity, even though we don't tie them together with a single object or variable

However, that's, of course, a pretty contrived way of doing it. In practice, we wouldn't spell out the state transitions by hand like this. We still want domain specific methods like "have birthday" that explain *why* a state is changing. Further, we want to show that the state we're producing logically follows from the current one. We want to be able to causally tie those two pieces of data together to cement their identity relation. This gives rise to a programming style that favors *computing* or *deriving* new states from existing data.

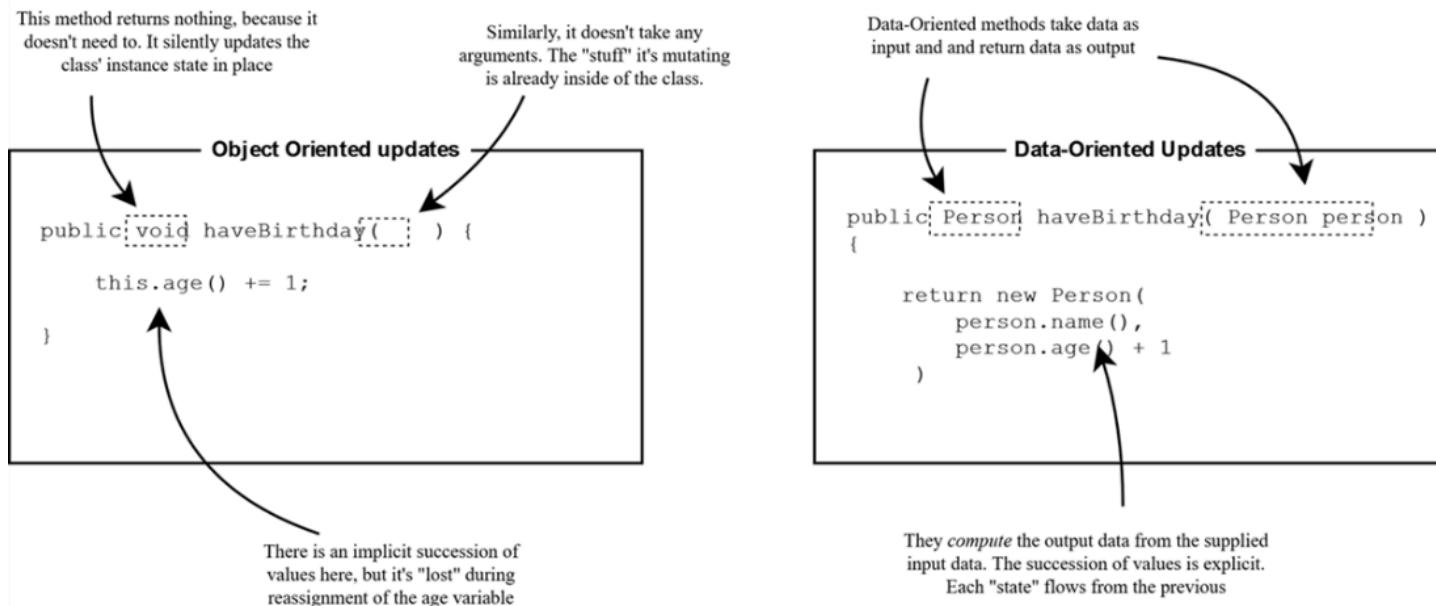


Figure 2.5 Comparing two method styles

We model change by writing methods that take data as input (reflecting the current "state" of our entity) and *return* new data as output (the next "state" of our entity).

Listing 2.29 Computing the next state from the current one

```
Person person = new Person("Bob", 32);
Person olderPerson = haveBirthday(person);      #A
// [out] Person(name="Bob", age=33);
Person evenOlder = haveBirthday(olderPerson); #A
// [out] Person(name="Bob", age=34);
```

#A We model the person getting older by creating a succession of values. The previous state is used to compute the next, which is used to compute the one after that, and after that, and so on

In doing so, we're able to create change in our program without ever actually changing anything. Outputs feed inputs, one after the other. We manage the change explicitly by plumbing the state from method to method. This is a subtle change in small examples, but transformative to our programs as a whole. It gives rise to a very different way of putting our programs together (we'll explore the full impact of this in later chapters).

There are other ways to achieve this outputs-feed-inputs chaining. In Figure 2.5, we modeled `haveBirthday` as a stand-alone static method which took data as a concrete input. However, we have some wiggle room on where and how we define these methods. For instance, rather than defining them on their own as static methods with explicit inputs and outputs, we can optionally define them directly on the value object itself, and pass the input *implicitly*. It's still "data in; data out," but the "in" part is handled transparently behind the scenes.

Listing 2.30 Another way of modeling immutable successions of values

```
final class Person {
    final String name;
    final Integer age;

    public Person haveBirthday() {                      #A
        return new Person(this.name(), this.age() + 1); #A
    }                                                 #A
}
```

#A Note that this method is very different from the object-oriented one despite living on the class. It doesn't modify any internal state (values cannot change!). It computes and returns a new value

This has a superficially similar look and feel to object-oriented programming, but it is still firmly in the world of data processing. The method *uses* the state of the object, but it doesn't *modify* anything on the object. Because, of course, it can't modify anything. The method is attached to a value class. It can only use the current state to compute the next one and return it as data.

Listing 2.31 Comparing different programming styles

```
Person person = new Person();          #A
person.haveBirthday()                 #A
// [out] Person(name="Bob", age=33);    #A

Person person = new Person("Bob", 32);   #B
Person olderPerson = person.haveBirthday(); #B
System.out.println(person)             #C
// [out] Person(name="Bob", age=32);    #C
System.out.println(olderPerson)         #C
// [out] Person(name="Bob", age=33);    #C
```

#A Updating an object identity in place

#B Superficially, it looks exactly the same as updating the identity in place, but it doesn't update anything at all!

#C The original person is still exactly the same and unmodified. It was just a convenient jumping off point for creating a new value

This again leads to its own unique coding style. Each method returns a new version of itself, which means we get fluent-like APIs for free. We can chain successive method calls together to build up larger results.

Listing 2.32 Fluent APIs for free

```
Person original = new Person("Bob", 32);
Person updated = person
    .haveBirthday()          #A
    .haveBirthday()          #A
    .haveBirthday();         #A

assert updated != person      #B
```

#A Fluent-style API's "for free." Each function returns a new value, which can be chained

#B But make no mistake, nothing is being modified here!

If we try to do similar chaining with stand-alone methods it's decidedly less elegant and readable.

Listing 2.33 Chaining method calls without a fluent API

```
haveBirthday(haveBirthday(haveBirthday(person)));    #A

Function.<Person>identity()                      #B
    .andThen(Chapter02::haveBirthday)               #B
    .andThen(Chapter02::haveBirthday)               #B
    .andThen(Chapter02::haveBirthday)               #B
    .apply(p);                                     #B
```

#A We have to chain calls "by hand" by explicitly feeding the output of one into the input of another

#B Or we have to pull in some additional scaffolding to enable chaining. It's not bad once you're used to it, but it's undeniably clunky by comparison

So, which is better? It depends. Different kinds of programs benefit from different styles (and different people prefer different styles). I've programmed extensively with both approaches (regularly mixing and matching in the same project) to great effect. Broadly speaking, I find math-y value-like things tend to benefit from having the methods defined directly on the body of the object. Their math-y nature often means that we

need to stack many related operations back-to-back to build up a new result. The fluent APIs make it easy to chain these complex operations while maintaining readability.

Listing 2.34 Readable vector transformations

```
Double result = new Vector(2, 3)
    .scale(10.0)          #A
    .subtract(new Vector(1, 1)) #A
    .magnitude()          #A

Double result = magnitude(subtract(scale(new Vector(2, 3), 10.0), new Vector(1,1))) #B

#A The more math-y something is, the more it tends to benefit from this style (broadly speaking)
#B It turns into a big mess if we try to do it with manual chaining of standalone functions
```

On the other hand, (continued heavy emphasis on the “it depends” and “just my opinion”) data tends to fall into the opposite bucket. I like to keep domain behaviors and operations completely separate from the data. That might sound strange from a traditional object-oriented perspective, but we’ll explore the rationale behind this in detail over the next few chapters. Drawing a line in the sand gives us some breathing room to focus on what the data means and how we represent it in code. The more time we spend understanding it, the more everything else tends to fall into place.

2.5 Modeling Data and Values with Records

Java records are the main tool we’ll use throughout the book for creating value and data classes.

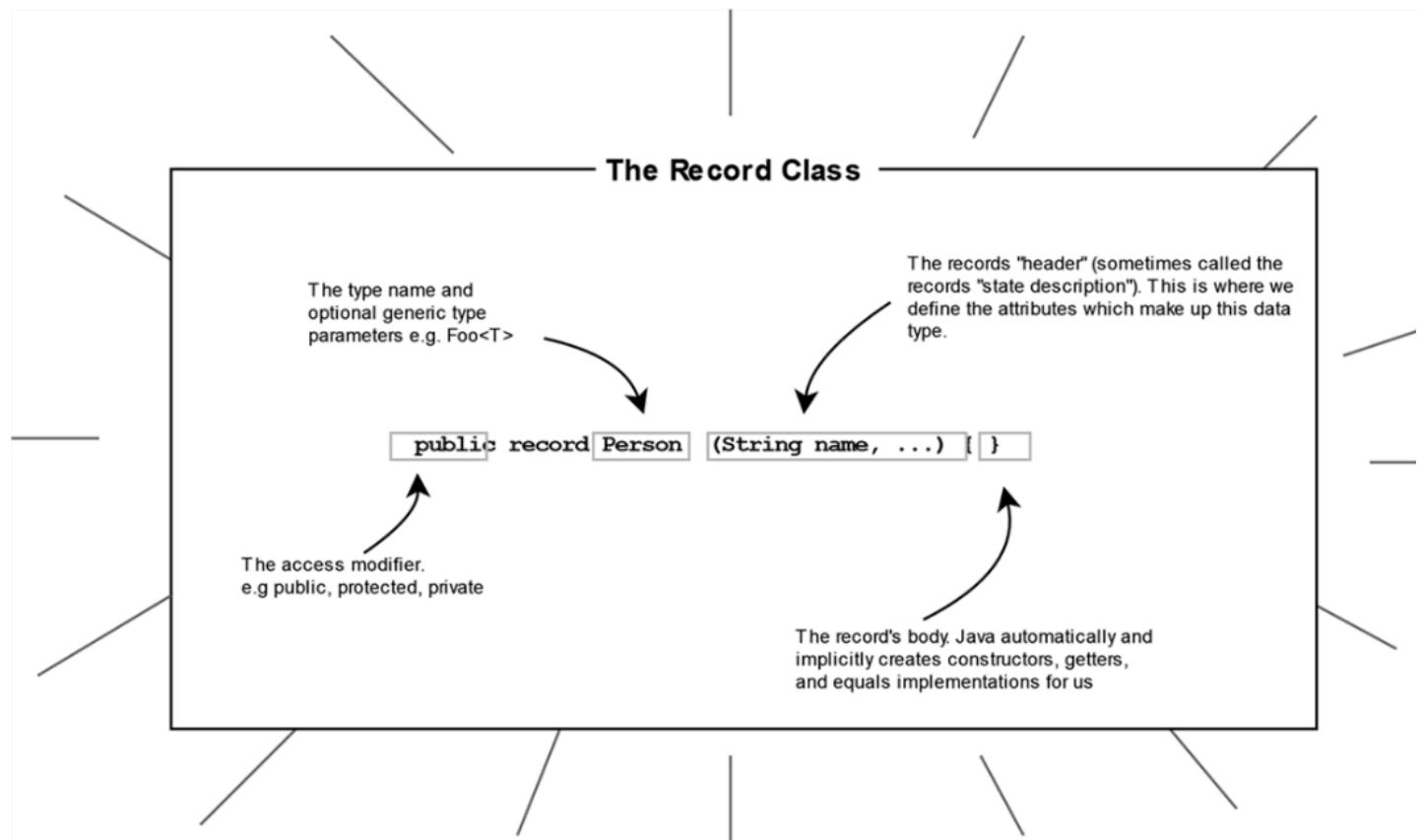


Figure 2.6 The record class

Records play a special role in Java. Superficially, it's alluring to categorize them as a boilerplate reduction tool (which is undeniably awesome), but their true purpose is semantic. They exist to be transparent carriers of values. They're tools for creating *value objects*. To support this style of modeling, they come with some constraints that traditional classes do not have. Most notably is that records do not support information hiding. You cannot decouple the internal representation from the external one. With records, what you see is what you get. They're defined completely and entirely by their *state description*, the attributes we specify in the header.

In exchange for this constraint, records give us a powerful and terse way to create new data and value types. They have all the properties we'd expect a good value object to have. They're effectively immutable (as much as they can be in Java) and, while they technically have an object identity (as all objects in Java do), that identity is not a semantically meaningful one. What a record *is* is determined by the single state we give them during construction. If the values they represent are the same, the records are the same.

Listing 2.35 Recreating the Vector type as a record

```
record Vector(double x, double y){}          #A  
  
final Vector a = new Vector(2, 3);           #B  
final Vector b = new Vector(2, 3);           #B  
final Vector c = new Vector(2, 3);           #B  
  
assert a.equals(b)                         #C  
  && b.equals(a)  
  && a.equals(c)
```

#A Recreating our Vector implementation from listing 2.7, but now using a record

#B Like good value objects, what they are is determined by their equality

#C Any value can be substituted for any other value with no observable change in behavior

Records have all kinds of interesting features and constraints, but I'll spare you my recreating the docs here. Instead, we'll take a quick tour through the relevant features we'll use throughout the book.

2.5.1 Compact Constructors

We'll make heavy use of the record's Compact Constructor. This works just like a traditional constructor, but it lets us skip a lot of the usual ceremony involved so we can focus on the more important details. This means that we don't have to specify the arguments (they mirror what's defined in our record's header) or assign them to instance variables.

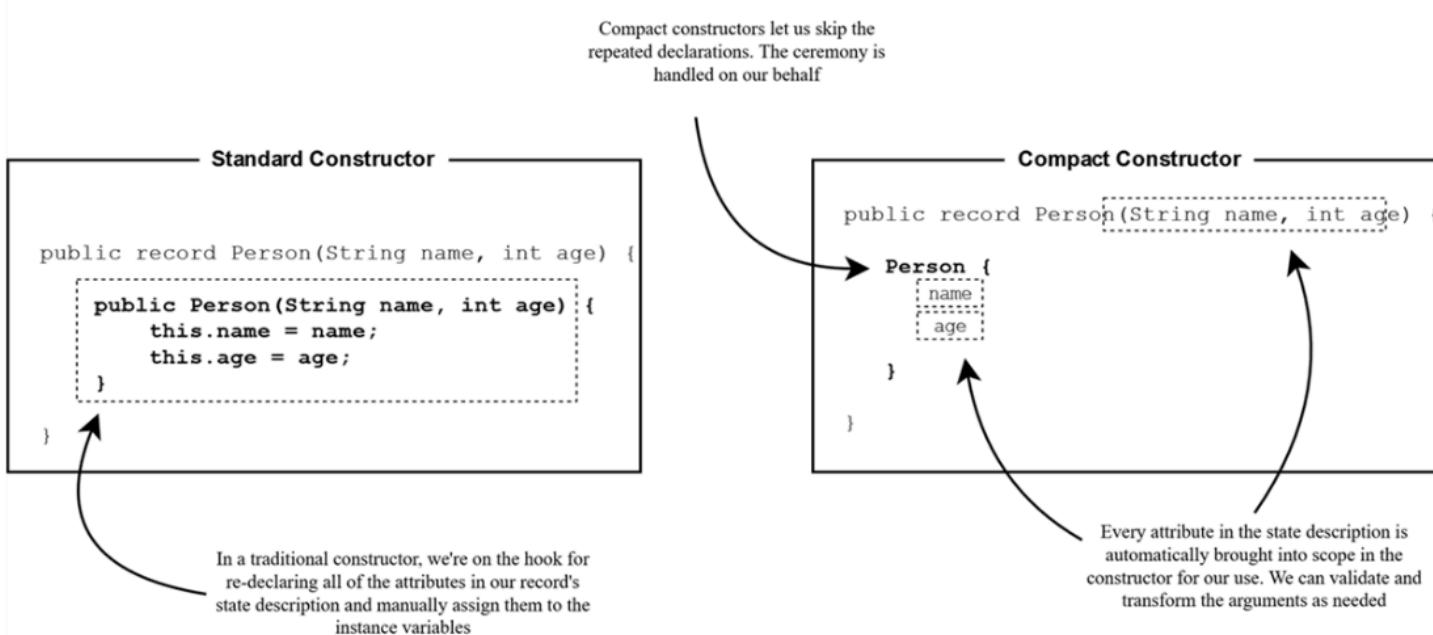


Figure 2.7 Comparing traditional versus compact constructors

The main reason we'll lean so heavily on compact constructors for is validation. Records are transparent carriers of exactly what we give them. If we give them a bunch of junk values, they'll be made up of a bunch of junk values. We don't want garbage data flowing through our programs, so the constructor is where we enforce the invariants.

Listing 2.36 Enforcing invariants during construction

```
record Rational(int num, int denom) {
    Rational {
        if (denom == 0) { #A
            throw new IllegalArgumentException( #A
                "Hey! That's not how math works!" #A
            );
        }
    }
}
```

#A Checking during construction makes sure that only valid data is created.

What kind of validation we put in the constructors is very specific in data-oriented programming. We don't use it as a dumping ground for various ad-hoc business rules. We use it to enforce what the data *is* – its semantics. Validation is how we guide everyone who uses this data type towards the right usage, even if they might not know what “right” is (for instance, new to the team / codebase / domain). Validation is a tool for communication as much as it is correctness. It's how we ensure the integrity of our data type even as the hands which work on the codebase change over time.

After validation, the other compact constructor feature we'll make heavy use of is the ability to transform the incoming arguments before their assignment.

Listing 2.37 Transforming arguments during construction

```
record Rational(int num, int denom) {  
    Rational {  
        if (denom == 0) {  
            throw new IllegalArgumentException(  
                "Hey! That's not how math works!"  
            );  
        }  
  
        int gcd = gcd(num, denom);          #A  
        num /= gcd;                      #A  
        denom /= gcd;                   #A  
    }  
}
```

#A While we're inside of the constructor, we can transform the incoming argument as much as we want. Here we're normalizing them before assignment.

This can be used to simplify incoming arguments (as in listing 2.37 above), but one of the main ways we'll put this to use in the book is by sanitizing those incoming arguments of any unexpected mutable references. Mutability is the destroyer of values. In fact, mutability is one of the main things we have to watch out for when programming with records.

2.5.2 Things to watch out for with records (Part I): shallow immutability

Records are only as value-like as the value objects we give them. A single mutable reference is all it takes to undermine the record's ability to model immutable data. As such, creating records requires care. This is especially true when the standard Java collections are involved.

Listing 2.38 Be careful with the standard Java collections

```
record Person(  
    String name,  
    List<String> friends  #A  
) {}
```

#A The value-ness of this entire data-type depends on the people who use it remembering to give an unmodifiable version of the List interface.

The value-ness of the Person class in listing 2.38 depends entirely on the specific List implementation that we give it during construction. A minor slip up, or misremembered API, and we drag our record back into the world of change and identities. Values that suddenly change are deeply jarring on a cosmic level. Change *shouldn't* be possible ("four is four!"). Change isn't defined for values. And yet, without care, that change can sneak in right under our noses.

Listing 2.39 Poisoning records with mutable references

```
List<String> friends = Arrays.asList("Joe", "Jane"); #A
Person person = new Person("Bob", friends); #B

System.out.println(person);
//[out] Person2[name=Bob, friends=[Joe, Jane]]

friends.set(0, "Billy"); #C
System.out.println(person); #C
// [out] Person2[name=Bob, friends=[Billy, Jane]] #D
```

#A Whoops! Accidentally called the wrong list constructor
#B And now a mutable list is hidden inside our “value”
#C Anyone that holds a reference can reach inside of the “value” and change its state
#D This should be impossible!

This is the type of “semantics” problem for which we’ll leverage transformation in the constructor. To *be* a value, we have to be built *from* values. We can enforce that during construction by creating our own copy of the (potentially mutable) incoming list. This guarantees our data is created with the precise value semantics we need.

Listing 2.40 Using the compact constructor to enforce invariants

```
record Person(String name, List<String> friends){
    Person {
        friends = List.copyOf(friends); #A
    }
}
```

#A The copyOf method produces a new UnmodifiableList behind the scenes. This guarantees we always see a value-based view of the collection, rather than a mutable one.

This small change has a powerful effect on the code as a whole. Now it doesn’t matter if somebody accidentally passes the wrong kind of list to our data type. We can’t be “poisoned” with mutability. We’re able to defend and keep our value-semantics regardless of what kind of collection we’re handed.

Listing 2.41 Anti-poison technology

```
List<String> friends = Arrays.asList("Joe", "Jane");
Person person = new Person("Bob", friends); #A

friends.set(0, "Billy"); #B
friends.set(1, "Michael"); #B

System.out.println(person); #C
// [out] Person2[name=Bob, friends=[Joe, Jane]] #C
```

#A Inside of our constructor, we’ve defended against this common mistake
#B Out here, they can freely mutate their collection as much as they want
#C But we remain an unchanging, immutable value.

However, despite this improvement, there’s still something a bit “off” with it. We’re communicating something to the reader that’s misleading. What we want that collection to be – a value – is hidden away inside of the constructor as a private transformation. We “know” it’s in there (because we just wrote it), but to everyone else, this looks like any other chunk of code that expects any Java Collection. The types we’ve chosen while modeling this code strongly conveys to the reader that mutable flavors of List are

allowed. It's like a little prank we're playing on the people who read our code. The code says one thing to them via the type system, "anything that implements List is A-OK!" but secretly means something completely different, "I am actually only cool with Unmodifiable lists." This gap is a little unexpected bump hidden in the code for them to eventually trip over.

Listing 2.42 An easy mistake to make, because the code is lying to us about what it allows

```
List<String> friends = Arrays.asList("Joe", "Jane");
Person person = new Person("Bob", friends);

person.friends().add("Billy"); // ERROR! #A
```

#A As far as what the code tells us, it looks like this should totally work, but anyone who tries will end up with an unexpected runtime error

What'd be nice is if we had a data type that let us inform the people reading our code of exactly what *kind* of list we want – an immutable one.

This is where the expressivity of records shines. They let us create small wrapper types with just a few lines of code. These types enable us to be really clear about the semantics we have in mind. In the case of the vanilla List interface being a bit ambiguous, we can introduce a new record that communicates exactly what *kind* of list we'd like.

Listing 2.43 Creating immutable list types

```
record ImmutableList<A>(List<A> items) {
    ImmutableList {
        items = Collections.unmodifiableList(items); #A
    }

    public static <A> ImmutableList<A> of(A... items) { #B
        return new ImmutableList<>(List.of(items)); #B
    }
}
```

#A We can give ourselves a semantically meaningful wrapper type in just a few lines of code.

#B And because records are just classes behind the scenes (more or less), we can give ourselves some nice Quality of Life features that simplify usage.

We've "spent" a few lines of code, but what we get back in exchange is a new level of semantic precision in our programs. We can use this data type to communicate what our designs *mean* – that we're dealing with immutable values.

Listing 2.44 Updating the person record

```
record Person(  
    String name,  
    ImmutableList<String> friends #A  
){}  
  
new Person(  
    "Bob",  
    ImmutableList.of("Jane", "Joe") #B  
);
```

#A Now there is no way to accidentally provide the wrong kind of collection. We guide everyone who uses this class towards the right path

#B We HAVE to provide an ImmutableList. Anything else would fail type-checking.

2.5.3 Things to watch out for with records (part II): nulls

We've done a lot of work in this chapter getting down to the heart of how we represent values and data using the tools available to us in Java. However, because we're in Java, a certain pesky part of the language is always lurking silently in the background waiting to undermine all of our modeling efforts at a moment's notice: null.

Listing 2.45 Creating a piece of "data" that consists entirely of nulls

```
Person person = new Person(null, null); #A  
  
#A A garbage piece of data that consists entirely of nulls
```

Nulls are a problem in the data-oriented view of the world because they rob us of our ability to trust our representation of the data in code. If something says it's a String, then it should *be* a String. It should *always* be a String. And it should never *not* be a String. Nulls undermine this very, very basic contract. If something in our domain is only available sometimes, then we want to make that optionality explicit in our data model. That optionality is valuable information about our domain. Nulls don't communicate anything, because they're implicit bypass mechanisms that side-step our modeling efforts.

So, the question here is: what do we do about it (if anything)?

MORE CONSTRUCTOR VALIDATION?

One obvious approach here would be to leverage more validation in the constructor. That's where we enforce the semantics after all. If nulls break those semantics, then we should filter them out.

Listing 2.46 defending against nulls in the constructor

```
record Person(String firstName, String lastName) {  
    Person {  
        Objects.requireNonNull(firstName); #A  
        Objects.requireNonNull(lastName); #A  
    }  
}  
  
Person person = new Person(null, null); // ERROR #B
```

#A Using the compact constructor form make sure nothing is null before allowing creation
#B This ensures that we explode with a stack trace rather than silently propagating nulls through our system (potentially causing much more harm)

However, I feel about explicitly defending against nulls during construction the same way I feel about explicitly writing out the `final` keyword everywhere: it can be a useful tool in the beginning, but can optionally be dropped depending on the team's preference and experience level. Programming is a constant balancing act between safety, expressivity, and boilerplate. Sometimes we'll give up one to achieve more of the others. Nulls are one of the areas where I generally choose to give up explicit checking on the records in favor of improved visibility into the more important semantic validation code. "This shouldn't ever be null" is treated as a given.

Further, throughout the book, we'll learn techniques for evicting nulls from our programs. The more powerful we become in our ability to use data modeling to control what can be expressed, the less of an issue these little gaps where nulls can sneak through become. When paired with tests, which we'll talk about later in the book, Null Pointer Exceptions effectively become a problem of the past.

2.5.4 Some other niceties that come with records

The expressiveness of records is pretty transformative in a language like Java. What used to take pages of code can now be done in a few lines. While records aren't "about" reducing boilerplate, the effect they have on boilerplate is a darn nice side effect.

Programming with records is sort of like putting on glasses for the first time after years of thinking that street signs were just printed blurry for some reason. The core stuff our programs operate on is laid bare -- unshackled by all the noise that used to surround it. You can start to put data types in places where you previously wouldn't have, and even physically lay out your code in ways which previously would have been awkward or visually noisy.

LOCAL RECORDS AND CLASSES

It's super common to need to stitch a few pieces of data together locally in the scope of a method (especially while writing tests). Previously, because creating classes in vanilla Java was expensive in terms of boilerplate, this was usually a scenario where we'd end up sacrificing names and readability in exchange for less tedium while writing the code. The main weapons of choice here would be tuples, pairs, or other positional based data structures.

Listing 2.47 combining results with positional data structures

```
Optional<Person> mostPopular(List<Person> people) {  
    return people.stream()  
        .map(person -> Tuple.of(  
            person, #A  
            resolveFriendGraph(person, people))) #A  
        .sorted((p1, p2) -> Double.compare(  
            p1.second(), #B  
            p2.first())) #B  
    )  
        .map(popularity -> popularity.first()) #C  
        .findFirst();  
}
```

#A Creating a Tuple that pairs together our Person along with their friend Graph

#B Accessing the contents of the tuple by position

#C More positional access. The further away we get from the tuple's origin, the harder it becomes to keep track of which slot holds what

But positional based tuples are less than ideal in practice ("What the heck is in `first()`?").

This positional problem gets worse the more data you need to stitch together and the more slots you need to fill ("What the heck is `fifth()`?").

This trade off changes when creating new data types is almost as syntactically lightweight as creating tuples or lists. It opens us back up to using data with explicit names rather than opaque positional accessors. Records are terse enough that we can define new data types locally inline right where we need them:

Listing 2.48 Defining records locally in a method

```
Optional<Person> mostPopular(List<Person> people) {  
    record Popularity(Person person, Integer totalFriends){}; #A  
  
    return people.stream()  
        .map(person -> new Popularity(  
            person,  
            friendGraph(person, people))  
        )  
        .sorted(Comparator.comparingDouble(Popularity::totalFriends).reversed()) #B  
        .map(Popularity::person)  
        .findFirst();  
}
```

#A Declaring a new record type right where we need it.

#B Now we can use clear names where we previously would have used opaque tuples or lists

Being able to tersely create data types right where we need them is a game changer for code readability. Humans avoid friction, even if small. Being able to quickly crank out well modeled and well named data wherever we need it massively lowers the cost of us being explicit and precise.

CODE LAYOUT

The increase in expressivity that records bring to the table can even change some of our core strategies for how we layout the code in our project. Broadly speaking, Java generally operates under the loose guideline of one class per file. This is totally sane

guidance in the world where a single class might require dozens of lines of code just to be minimally viable. Keeping multiple classes in the same file is often just too visually noisy. However, when we can tersely express our data types in one or a few lines, we gain a lot of wiggle room to decide where we want to store things. It can start to make sense to store related data types together in the same file.

Listing 2.49 Core data types for a card game in 4 lines

```
interface CardGame {  
    enum Suit {Hearts, Diamonds, Club, Spade;}  
    enum Rank {ONE, TWO, THREE, FOUR, FIVE, /*...*/;}  
  
    record Card(Rank rank, Suit suit){};  
  
    record GameState(List<Card> drawPile, List<Card> discards){}  
}
```

This small change can be a massive boon to program understandability. It's hard to overstate the joy of dropping into a program and being able to see all of its core data types in one easily digestible view rather than fractured across multiple files. Stacking related data types together like this is a technique we'll use throughout the book. Java's latest versions have some fun things in store for this kind of modeling!

2.6 What to do if you're on a JDK older than 17

The older versions of Java hold a very special place in my heart. Large corporations have no shortage of legacy code bases that are still rocking versions of the JDK all the way back to 8 (or older!). Despite their growing age (the current JDK is up to 23 at the time of writing), I still find myself working with older versions of Java regularly. I assume that a fair number of people reading this book will find themselves in the same slow-moving boat.

The good news is that, despite the lack of modern constructs like records, data-oriented programming is still both possible and comfortable in the older versions of Java. The latest tools are nice, but they're just tools.

So, what do we do?

The first option is to tough it out. Use vanilla classes to model data like we did in section 2.2. It's verbose, and tedious, but it gets the job done. Plus, most modern IDEs can generate the boilerplate for us with a few keystrokes.

The other option, and the one I always opt to go with, is to pull in a 3rd party annotation processor that can generate the boilerplate behind the scenes. Annotation processors are incredibly powerful, but, as a heads up, just know that they evoke very strong opinions from developers. Few people love them. Many more hate them. I find that that their usefulness far outweighs their downsides.

There are a few major players in the scene, but two of the more popular ones I've used are Lombok (projectlombok.org) and Immutables (immutables.github.io). Both are annotation processors that can turn "incomplete" class definitions into fully fledged value classes (with all the methods we need). They differ slightly in how they accomplish this (Lombok via byte code manipulation; Immutables via source

generation), but they both get us to the same end state: easy record-like modeling of data.

I'll spare you my recreating the full documentation for each library here, but as a brief taste of their power, we can look at how the Person record we've been using throughout section 2.6 would look when using these annotation processors.

Listing 2.50 Lombok annotated class

```
import lombok.Value;

@Value #A
public class Person {
    String name;
    Integer age;
    #B
}

Person person = new Person("Bob", 32);    #C
Person person = new Person("Bob", 32);    #C
person.equals(person);                  #D
```

#A The value annotation creates a constructor, marks our fields final, generates getters. The whole thing.

#B Similar to records, this body is blank because the annotation processor handles everything

#C The constructor shows up for free.

#D Value annotated classes are proper value classes. They behave like values.

Here's the same thing, but using the Immutables library.

Listing 2.51 Immutables annotated class

```
import org.immutables.value.Value;
@Value.Immutable
interface Person {                      #A
    String name();
    Integer age();
}

Person person = ImmutablePerson #B
    .builder()                 #C
    .name("Bob")
    .age(32)
    .build();
```

#A The two frameworks have very different approaches. Immutables' annotations are attached to interfaces or abstract classes

#B And unlike Lombok, it generates a totally new class. We use that class when instantiating the data types

#C The other big departure is that Immutables doesn't generate a public constructor by default. It generates a builder.

How cool is that? They enable us to model data in older Java versions with minimal ceremony and boilerplate. If

you're still hacking away on a project that's not able to upgrade to newer JDK, definitely consider pulling in one of the popular annotation processors. They do a great job of closing the capability gap between classes and records.

2.7 Wrapping up

In this chapter, we looked at the many forms of identity that we encounter while programming. Object and variable identities are the main standbys of OOP. However, we also learned that these are just one of the many ways we can model the *idea* of identity. The person is not the attributes we use to describe them. Identity is a choice we get to make while designing.

We also looked at values, which have no notion of identity. Values just *are*. They don't change and *cannot* change. Further, we learned how to represent values in Java with value classes. These classes are immutable and defined entirely by the (single) state which they hold.

Once values were under our belt, we were able to look at a precise definition for data. Data is what values become when we make them *about* something. The integer 4 as a value just *is*. It's not data. But make that same Integer about something, like a temperature, age, or voltage, and it *becomes* data. It's now modeling a domain. With that comes a semantics which makes that data *that data*.

Lastly, we looked at records. These are Java's modeling tool for creating data and value classes. They let us create new types with minimal ceremony. We took a quick tour of the relevant feature's we'll use throughout the book. The new compact constructor is our main tool for validating and transforming arguments on the way in to ensure the semantics of our data model remain intact.

Next up, we'll dive into how data-oriented programming works. It's a deep dive into what it means for data to "have a semantics." This gets us to the essence of data-oriented programming: representation.

2.8 Summary

- Identity gives continuity to a related set of values as they change over time
- Object identity is the most tangible in Java. It shapes "how" we usually program.
- Variables create an identity that associates values over time with a named location
- Values just are. They transcend our programs. They're immutable and unchanging.
- Values have no notion of identity.
- Value *objects* are how we represent values in Java
- Value objects carry the same properties as the values they model: immutable and defined by equality
- A value assigned to a non-final variable is not a value (though, we ignore this and manage by convention)
- All Java collection types can be used to model values. It just requires care to ensure the correct semantics
- Data "as data" means modeling data as values
- Values become data when we give them meaning or context.
- Data is more than collections of facts. It can model abstract ideas, or decisions, or state itself

- Data can model identity and change over time. We model this by creating a succession of values
- Records are modeling tools for creating value and data objects
- Records have a compact constructor, which lets us skip a lot of the ceremony of traditional constructors
- The validation we put in record constructors guards the value's semantics
- Transformation of incoming constructor arguments can be used to enforce specific value semantics
- The terseness of records enables us to introduce new semantic layers into program with minimal overhead
- The terseness of records also allows us to leverage new code layout strategies.
- Annotation processors like Lombok and Immutables can generate value classes on older JDKs

3 Data and Meaning

This chapter covers

- Exploring data's meaning
- Moving from meanings to constraints
- Capturing it all in code

In the previous chapter we explored what data *is*; now we're going to explore what it *means*. It might sound a bit abstract at first, but this pursuit of "meaning" is one of our primary tools for building rock solid systems. To quote Leslie Lamport, "It's a good idea to understand a system before building it". Being very explicit about what our data *means* is how we get down to that fundamental level of understanding prior to any coding.

Once we know what our data means, we have the much simpler task of moving that meaning into code. We'll explore various techniques we can use for data modeling. We'll look at which ones work, which ones fall short, and the criteria we can use for weighing if our modeling is moving in the right direction or not.

The last thing we'll do is turn all this talk about meaning towards the type system itself. The types we use while modeling our data can have a significant impact on the meaning of our code as a whole. They determine how easy or hard it is to do the right thing. We'll learn to see types under a new light and visualize them for what they are: sets of values. Once we can do that, we unlock a powerful form of analyzing our modeling in order to squeeze out potential bugs before they're allowed to surface.

Let's dive into the world of meaning!

3.1 “What does it mean?”

The process of figuring out what the data in a new domain means can be a daunting challenge. This is true even for an experienced developer. The ever-present problem is that you don't know what you don't know. So, what questions do you ask? Where do you start? How do you know when you're done? If you try to solve everything at once, it's easy to get overwhelmed and have your mind go blank as you stare into the fog of war. So, a good way to combat the complexity is to just start with the most basic and fundamental question you could possibly ask: "what is this thing?"

That's the whole process. For any individual idea or concept in our domain, we can get started by picking some arbitrary aspect of it at random, and, asking "what does *that* mean?". And then, like a curious and slightly obnoxious child, *continue* asking that same question over and over again until we've reached bedrock (or our audience loses patience). Simple questions, pursued thoughtfully, produce powerful results.

To explore this, we're going to walk through the process of modeling a small piece of data from one of my favorite topics, and one I force into any conversation when I've got

somebody cornered (which is now you, reading this): how a class of woodworking finishes, called drying oils, cures over time.

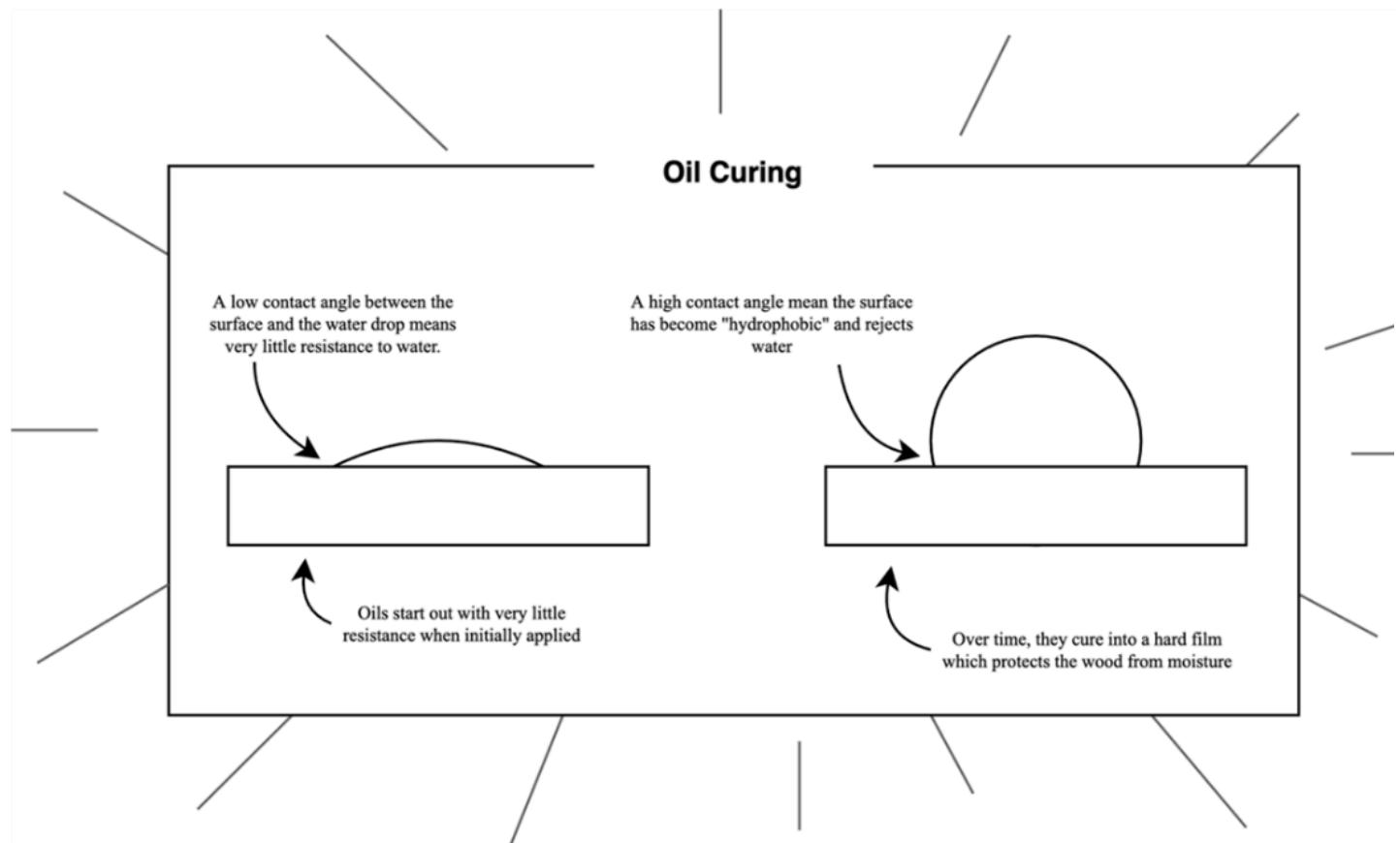


Figure 3.1 The basics of drying oils

When these oils interact with oxygen, they undergo a chemical reaction which converts them into a hardened film which protects the wood from moisture. This reaction process is impossible to see with the naked eye, but it can be measured indirectly by testing how much the wood has begun to resist water. And we can measure that by the angle a water droplet forms when placed on the surface. The larger the angle, the more the finish has cured. The oil is done curing (enough) when the contact angle stops changing day to day. How interesting!

Here's a sample of one of measurements in JSON. We're going to convert this into Java.

Listing 3.1 Some (very interesting) data on Tung oil curing behavior

```
{  
  "sampleId": "UV-1",  
  "day": 3,          #A  
  "contactAngle": 17.4 #B  
}
```

#A Tung oil cures painfully slowly in open air. Full days are about the smallest granularity at which progress can be tracked

#B The contact angle formed between a droplet of water and the surface

To model this in Java, we might start with the usual “code-first” developer approach of jumping straight into our IDEs to begin some finger typing. This kind of approach is fun, and feels pretty productive, because it’s largely a translation process that doesn’t

involve much (if any) thought. We mechanically map the “shapes” we see in the data one-to-one (or as close as we can get them) to appropriate types in Java. So, text-like things in JSON become Java Strings in our code. Number-like things become Java ints or doubles. The result of that process would likely look something like this:

Listing 3.2 A mechanical translation from JSON to Java

```
record Measurement(  
    String sampleId,          #A  
    int daysElapsed,         #A  
    double contactAngle      #A  
) {}
```

#A We picked types which best mirrored the incoming JSON types (numbers, strings, etc)

Which has the right overall shape, but the important data-oriented question is: does it have the right meaning? To figure that out, let’s take a step back from the details of the code and just focus on what the data itself means. Once we’ve done that, we’ll be able to better address if we’ve captured that meaning in our model.

Listing 3.3 Going back to the data

```
{ "sampleId": "UV-1",      #A  
  "daysElapsed": 10,        #A  
  "contactAngle": 6.28     #A  
}
```

#A Each of these fields has a rich meaning associated with them that we need to capture in our code

So, we’ll kick off the process by picking a random field to investigate: daysElapsed. As we try to figure out what this data *is*, one of the first things that might pop up is that “days” is a pretty weird and ill-defined unit of time. What does that *mean*? 24 hours? The same time every day? Why not provide explicit timestamps? Instead, we just have “days”, whatever that means. If we’re going to accurately represent this in our program, that meaning is extremely important.

Now, for more fun oil curing background information: curing is painfully slow. Entire days are just about the shortest interval of time in which changes can be detected – and even that is pushing it. Because of this, the measurements (when performed by me, who is lazy) were done casually and spread out across the various samples every few days in order to avoid squinting and counting off multiple decimal places due to subtle changes in the contact angles.

That information might not seem super relevant at first, you might be dipping into “who cares?” territory, but this information is fundamental to building maintainable systems. By asking these questions we learned something about what our data means. Without this understanding, it would be very safe, but completely wrong (!), to assume that our data would be made up of daily measurements counting off every day that had elapsed. e.g. day 1, day 2, day 3..., but what we’ve learned is that the measurements are sparse. Seeing daysElapsed form a sequence like day 1, day 3, day 4, day 7... wouldn’t represent missing data, or a bug in our data ingestion, or any cause for alarm at all. We know that’s the way the data is. That’s how it was designed to be! The only way to get this understanding is by stepping outside of our programs.

Further, now that we know what it is, we know a lot about what it must not be. We've discovered invariants that our system must uphold in order to be correct. Days, being a measurement of time, can only represent that no time has yet passed ("0 days elapsed"), or that some time has passed (n days elapsed). However, "negative 15 days have elapsed" would make no sense at all. By pursuing what a piece of data means, we've simultaneously learned quite a bit about the invariants our system must maintain in order to be correct.

So, we take our refined understanding and write it down somewhere.

Listing 3.4 Writing down our understanding of daysElapsed

SampleID: Unknown

Days Elapsed:

A positive integer (0 inclusive).
Not strictly daily. Will be Sparse.

Contact Angle: Unknown

Let's keep going with the other attributes and turn our attention to `contactAngle` . This one is harder to pin down. What *kind* of angle is it? What is the unit of measurement here? Degrees? Radians? *Gradians*? Units are one of the most common things to mess up in programming. Not because they're inherently hard to deal with, but because they're so often left unspecified or rely on assumption of "the obvious" or "right" units to use. Our job is again understanding what this data means, which means putting on our detective hat and getting down to the bottom of it.

For this data, which is *my* data (you're still playing the role of someone cornered at a party while I drone on about wood finishing), these angles are all recorded in degrees (0-360) rather than radians. Further, we know from the investigation into days elapsed we just did, that curing is a slow and subtle process with very tiny changes over time. Which means that unlike the *rate* at which changes are tracked, which can be low fidelity, the measurement *of* that change must be highly precise in order to record the subtle shifts in angle. The rig I use (my phone) for capturing those contact angles is accurate to about 1/100th of a degree. Anything beyond that is too fuzzy due to limited resolution.

So, we'd again write down what we know somewhere.

Listing 3.5 Writing down our understanding of contactAngle

SampleID: Unknown

Days Elapsed:

A positive integer (0 inclusive)
Not strictly daily. Will be Sparse

Contact Angle:

degrees ranging from 0.0 (inclusive) to 360 (non-inclusive)
Precision of 100th of a degree

3.1.1 Discovering more about your domain

Let's finish up this exercise by taking a peek at the final attribute: `sampleId`. I saved this one for last because it's one of the most instructive in how a simple question like "what does it mean?" can lead you to discovering entire worlds hidden within your domain.

To start with, anything that gets called an "ID" should be immediately suspect. It is pretty unique to software developers as a group that "an id" means something opaque like a UUID or DB sequence number. Identifiers in the wild are often heavily steeped in domain information. Which is to say: they're like that for a reason. It's easy as programmers to fall into the trap of thinking that all IDs are as opaque as the ones auto-created by our systems. This mindset can cause us to overlook valuable domain information by lumping things under overly-broad "catch-all" definitions of "meaning" like this:

Listing 3.6 Incomplete domain analysis

```
SampleID:  
  Alpha-numeric + special characters  
  Globally unique.  
  
Days Elapsed:  
  A positive integer (0 inclusive)  
  Not strictly daily. Will be Sparse  
  
Contact Angle:  
  degrees ranging from 0.0 (inclusive) to 360 (non-inclusive)  
  Precision of 100th of a degree
```

The definition is *technically* "correct," but it's far from *precise*. The string "0FAA3B121DF4" meets the broad description of "being made up of characters", but if we stop there it robs ourselves of understanding what those characters *mean*. We'd miss that those characters represent, say, a Mac Address, and that there are semantics we can validate for what it means.

So: If we ask what it means to be a Sample ID in the domain of Chris' oil curing data, I'd tell you that Sample IDs are actually two things combined into one. (Not for any particularly good reason. That's just how I settled on scribbling them down for convenience.)

```
{  
  "sampleId": "UV-1",      #A  
  "daysElapsed": 10,  
  "contactAngle": 6.28  
}
```

#A IDs are made up of a curing method paired with a sample number

They follow the general scheme of "{method}-{number}", where "method" is the curing conditions under test, and "number" is a simple counter of each sample in that category. e.g UV-1, UV-2, UV-3, etc.

Now we understand the mechanics of that sample ID field. Do we stop here and write it down in our notes? It depends on what *kind* of understanding we're after.

It's worth taking a second here to refocus on the fact that at each step in this process we're specifically pursuing what something *is*. This is a very different process from simply analyzing how it happens to be *represented*. For instance, if we stopped probing into the meaning of the ID at this point, what would we actually know? The *schema* of the field? That it's in the form "{method}-{number}"? The constraints we've uncovered so far are all about how the sequence of characters are laid out. Effectively, we've established rules that we might feed into a lexer in order to tokenize the ID. We've discovered the syntax, but not the *semantics*. If we stopped now, we might write down our understanding like this:

Listing 3.7 Writing down our current understanding

```
SampleID:  
  A sequence of characters satisfying the regex /[A-Z]+-\d+/  
  Globally unique.  
  
Days Elapsed:  
  A positive integer (0 inclusive)  
  Not strictly daily. Will be Sparse  
  
Contact Angle:  
  degrees ranging from 0.0 (inclusive) to 360 (non-inclusive)  
  Precision of 100th of a degree
```

This definition is once again *technically* correct, but is it precise? Definitely not. It ignores all of the interesting stuff that those groups of characters *mean*. We've discovered this whole other side of our domain that needs understood! What *are* these curing methods? How many are there? We cannot say we've understood what it means to be a Sample ID without understanding the contents of which it is comprised.

So, we must dig down one more level. If we do so, we'd find that there are three different curing methods in the data: Air, UV, and Heat. These are invariants we need to enforce in our system.

With all of our new domain knowledge in place, that finally brings us to the close of our long journey. Let's write down our understanding of each component in the domain:

Listing 3.8 Our finalized understanding of the data

```
CuringMethod: #A
  One of: (AIR, UV, HEAT)

SampleNumber:
  A positive integer (0 inclusive) #A

SampleID:
  The pair: (CuringMethod, SampleNumber)
  Globally unique.

Days Elapsed:
  A positive integer (0 inclusive)
  Not strictly daily. Will be Sparse

Contact Angle:
  degrees ranging from 0.0 (inclusive) to 360 (non-inclusive)
  Precision of 100th of a degree
```

#A Totally new domain information we've uncovered

This is the information that was missing from our original model. There was a rich world of meaning behind each of those individual attributes. These are the invariants and constraints we expect our program to understand and enforce.

3.1.2 The importance of understanding meaning

This process, despite it not involving a single line of code, is the very core of programming to me. Software development is only in part about writing code or “shipping”. Above all else it’s about solving customer problems. If you want to do that effectively, you have to know how to do this work and ask these questions. You have to get down to the heart of what’s being talked about in a given domain. This is where the hard work is done: *the understanding*. The quality and trajectory of a software system is set here, during this slow and methodical process of questioning, long before any code or architecture gets laid down. It’s only in understanding what your data means that you can fit a software system around it.

3.2 Capturing the meaning in code

We’ve done the hard part of figuring out what our data means. Now we have the much simpler task of representing it in our code. Rather than start over from scratch, let’s use our new found knowledge of the data and its meaning to refactor the existing Measurement record we made at the beginning of this chapter.

Listing 3.9 Reminder of where we left off

```
record Measurement(
  String sampleId,
  Integer daysElapsed,
  double contactAngle
) {}
```

What we're missing from the current model is what this Measurement thing is supposed to *mean*. It has the right shape, but none of the essence. If we were dropped fresh into this code with no other context, would we know how to create data that's semantically correct? Would we know how to use it correctly in a computation? More importantly, would the code prevent us from doing the wrong thing when we didn't know what correct *meant*? If we take our current model as is, the answer is definitely no. We can freely plug in any values that the type system will allow whether they make sense or not.

Listing 3.10 constructing nonsense

```
new Measurement(UUID.randomUUID().toString(), -32, 9129.912); #A
```

#A Would this be valid data in our domain?

While this may be a well-formed program, it's definitely not a *correct* one. Each attribute we passed as an argument to the constructor violates one or more of the invariants we established for our data in the previous section. Good code should be self-describing and keep us from making silly mistakes. Let's try to make our code do that.

3.2.1 Detailed documentation and better variable names

One thing we might try is taking all the rules we figured out in section 3.1 and putting them into the source code as java doc on our record. It gives us a chance to lay out all of our assumptions about the data in one logical place. Also, it looks pretty professional.

Another thing we might do is put some of that documentation about our data's meaning directly into our variable names. For instance, we might rename `contactAngle`, which previously left us guessing at the units involved, to `contactAngleDegrees`, which removes all ambiguity by encoding the unit information right into the variable name.

Listing 3.11 Adding extensive documentation to our record

```
/**  
 * An individual observation tracking how water contact          #A  
 * angles on a surface changes as oil curing progresses by day #A  
 *  
 * @param sampleId  
 *     A pair (CuringMethod, positive int) represented          #B  
 *     as a String of the form "{curingMethod}-{number}"          #B  
 *     CuringMethod will be one of {AIR, UV, HEAT}              #B  
 * @param daysElapsed  
 *     A positive integer (0..n)                                #B  
 * @param contactAngle  
 *     Water contact angle measured in degrees                #B  
 *     ranging from 0.0 (inclusive) to 360 (non-inclusive)    #B  
 */  
record Measurement(  
    String sampleId,  
    int daysElapsed,  
    double contactAngleDegrees    #C  
) {}
```

#A a helpful high-level comment about what this record is modeling
#B Adding all of our domain information as param level documentation
#C Documenting the units directly in the variable name

Is this a step forward? I'd say so. People will be able to read our docs and use our class correctly. Plus, this level of documentation gives it a certain air of officiality. It has the appearance of being a well thought-out and well researched representation of oil curing measurements. However, we have to be somewhat careful here. It's easy to confuse the process of making the code *look* polished with the process of giving the code *meaning*. If we put that meaning into the code, it will help prevent us from making mistakes.

When we talk about making "mistakes", what we mean concretely is accidentally breaking one of our data's invariants – things which must be true at all times in order to be well-formed and correct (for example, `daysElapsed` must at all times only be zero or a positive integer, otherwise we'd say its invariant has been broken). Good code should be able to defend its own invariants and guide us towards correct usage.

Has our documentation accomplished this task? It probably depends on who reads it. Or if they've read it at all. Or what their mood was. Or if they were tired. Or... any number of other human factors that could cause them to make a mistake. At the end of the day, documentation lives *in* the code, but it is *not* the code. Even with the best intentions from the most documentation reading programmer of all time, the code itself still allows that well-intentioned documentation reading human to create garbage data.

Listing 3.12 breaking every single invariant at once

```
new Measurement(#A
    "1",      #B
    -12,     #C
    360.2    #D
);
```

#A Ignoring our documentation!
#B Not a valid sample id!
#C Not a valid count of days elapsed!
#D Not a valid measurement of degrees!

3.2.2 Enforcing constraints during construction

To enforce data's meaning and empower the code to prevent us from doing the wrong thing, we need to validate that it honors our constraints prior to getting created.

Where we put this validation logic is a big topic (one we'll dig into throughout the book). There's no easy answer, but for the sake of simplicity, and keeping the examples small, we'll keep all the validation logic on the data type itself.

So, since we've decided where we'll put it (for now), the next improvement we might try is taking all of the stuff we've currently got in our docs and turning it in concrete validation checks in the constructor.

Listing 3.13 Defending invariants during construction

```
record Measurement(  
    String sampleId,  
    int daysElapsed,  
    double contactAngle) {  
  
    Measurement { #A  
        if (!sampleId.matches("(HEAT|AIR|UV)-\\d+")) { #B  
            throw new IllegalArgumentException(  
                "Must honor the form {CuringMethod}-{number}" +  
                "Where CuringMethod is one of (HEAT, AIR, UV), " +  
                "and number is any positive integer"  
            );  
        }  
        if (daysElapsed < 0) { #C  
            throw new IllegalArgumentException(  
                "Days elapsed cannot be less than 0!");  
        }  
        if (!(Double.compare(contactAngle, 0.0) >= 0  
            && Double.compare(contactAngle, 360.0) < 0)) {  
            throw new IllegalArgumentException(  
                "Contact angle must be 0-360");  
        }  
    }  
}
```

#A Note the use of the record's compact constructor

#B Validating that the Sample ID is in the right shape

#C Validating the remaining constraints

If we take this approach for a spin, it feels pretty good at first. We've completely fixed the problems from listing 3.12 which allowed invalid data to be constructed. Now we can't create incorrect data even if we wanted to! The code prevents us from doing the wrong thing.

Listing 3.14 No more invalid data

```
new Measurement(  
    "1",      #A  
    -12,     #A  
    360.2    #A  
)
```

#A We can no longer instantiate invalid data. These will all throw exceptions if we try

This is the start of something interesting. We've baked information about what our data *means* directly into the code. Our type can enforce its own constraints even if the programmers using it failed to read the documentation or ignored its naming hints. We've used our data type to create a new layer of safety in our program.

However, there's still something slightly off here. What happens if we try to program with any of the data we just created?

```
double angle = measurement.contactAngleDegrees(); #A
```

#A Hm. A double. Like any other double.

What is this double? What's happening is that our data's meaning only lives ephemerally "inside" of the record during construction. Once the construction process is over, fields like our contact angle end up stored inside of our record as "just" a double. Of course, we "know" that that double represents angles measured in degrees, but as far as the code is concerned, it's just a double, like any other double in the system. We're right back where we were before: we need to explain what this double means through the use of comments or explanatory variable names. Further, we have to do this reasoning about what our code actually means *everywhere*.

Listing 3.15 Tracking meaning throughout a program

```
List<Measurement> measurements = List.of( #A
    new Measurement("UV-1", 1, 46.24),
    new Measurement("UV-1", 4, 47.02),
    // ...
    new Measurement("UV-2", 30, 86.42)
);

Map<String, List<Double>> bySampleId = measurements.stream(). #B
    .collect(groupingBy(
        Measurement::sampleId,
        mapping(Measurement::contactAngleDegrees,
            Collectors.toList())));
}

// Comparing the first and last samples in each
// group to see how much things changes while curing
List<Double> totalChanges = bySampleId.values() #C
    .stream()
    .map(x -> x.getLast() - x.getFirst())
    .toList();

// computing some summary stats
double averageChange = totalChanges.stream()
    .collect(averagingDouble(angle -> angle));
double median = calculateMedian(totalChanges);
double p25 = percentile(totalChanges, 25);
double p75 = percentile(totalChanges, 75);
double p99 = percentile(totalChanges, 99); #D
```

#A Up here at the top we know exactly what our data means.

#B But then it gets transformed. We have to pay close attention to understand that those doubles in the map represent Degrees

#C More transformations. More distance. Do these still represent degrees...?

#D By the time we're down here, what these doubles represent is very blurry. Still Degrees? The only way to understand this code is by working your way backwards through the call stack to mentally track how the meaning of the data changed as it was transformed.

Each time we assign data to a new variable, or transform it, or pass it to a method, something about its "it-ness" risks getting lost in translation. The further we get from where the data was originally defined, the harder we have to work to make sure its meaning gets transferred along with it. In an actual program (as opposed to tiny examples that fit on a page in a book), it presents a real challenge, because the meaning has to survive across multiple layers of abstraction, indirection, and boundaries. This lossy transfer of meaning creates a very real potential for bugs due to misuse stemming from innocent misunderstandings.

Then there's the matter of our *constraints*.

As soon as we leave the bounds of our Measurement type and get dropped back into the world of the double, all bets are off. Nothing prevents us from using or modifying our data in a way which doesn't honor what that double is supposed to represent.

```
double angleInDegrees = measurement.contactAngle() * 1000.00. #A  
double amountInUSD = 10.32
```

```
double whatTheHeck = amountInUSD + angleInDegrees; #B
```

```
double wrong = Math.sin(angleInDegrees); #C
```

#A Breaks our Degree invariant!

#B ...!?

#C Stop doing that!

What's going on here? Despite putting all kinds of defenses around our data type, adding fancy documentation, and explicitly naming our variables, we're still in this weird situation where all of that effort becomes moot whenever we read any data out of our record. What we want our data to *mean* seems to be at odds with what the types system says is *allowed*.

3.2.3 Obvious things which maybe aren't so obvious

Types have meaning just like data. When we pick a type like Integer, String, or Double we're communicating something very specific about the universe of discourse we want to take part in. This communication is also special. In addition to giving the humans reading the code valuable information, types communicate their meaning to the compiler itself. If we get the meaning of the type right, the compiler can help us enforce the meaning of the data it models. If we get it wrong, we experience the friction we were seeing in the previous sections, where our code constantly seems to be "forgetting" its own meaning.

How do we interpret a type's meaning, or, what do we mean by "mean?" There are actually a few schools of thought on this. However, the one I've found most useful for application development, is to mentally swap out the type's name for the *concrete set of values* that it denotes. So, if we were to look at, say, an `int` in Java, we'd "see" it in our mind as being made up of the finite set of integers from -2^{31} to $2^{31} - 1$ (the range of values that 32 bits can represent in two's complement (you can also find these numbers by looking at Integer's `MIN_VALUE / MAX_VALUE`)). We could further imagine "a type" as just being the name we give to that set of values:

```
int={-231,...,231-1}
```

Which, on the one hand, I mean, obviously, right? But on the other hand... well, maybe not so obvious? I programmed for a very, very long time before I started performing this mental swap in my head. It's extremely easy to pick and choose types as hand-wavy buckets of stuff without thinking about what those types *mean*.

This idea, as simple and "obvious" as it is, ends up being pretty transformative. Being introduced to this was like having a switch thrown in my head. When you start to see "through" the types into the sets of concrete values that they denote, it becomes very hard to not suddenly start noticing that a lot of the types we use in Java fail to capture

the essence of the domain idea we're trying to model. In fact, an alarming amount of the time our types denote sets of values which are directly contradictory to what we're saying they model!

This brings us back to that "friction" we were facing when dealing with our data model. Something about the types we've chosen has caused them to be at odds with what our data is supposed to mean. To figure out where we went wrong, we just have to do another simple exercise: compare the meanings.

3.3 The type system's effect on meaning

Comparing meanings gives us a powerful tool for "debugging" our modelling. We just have to look at the meaning of our data, and then look at the meaning of the type we've chosen to represent it, and then simply see if they agree with one another. If their meanings don't line up, that's a decent "smell" that our modeling might be off.

Let's do that exercise with the contactAngle attribute.

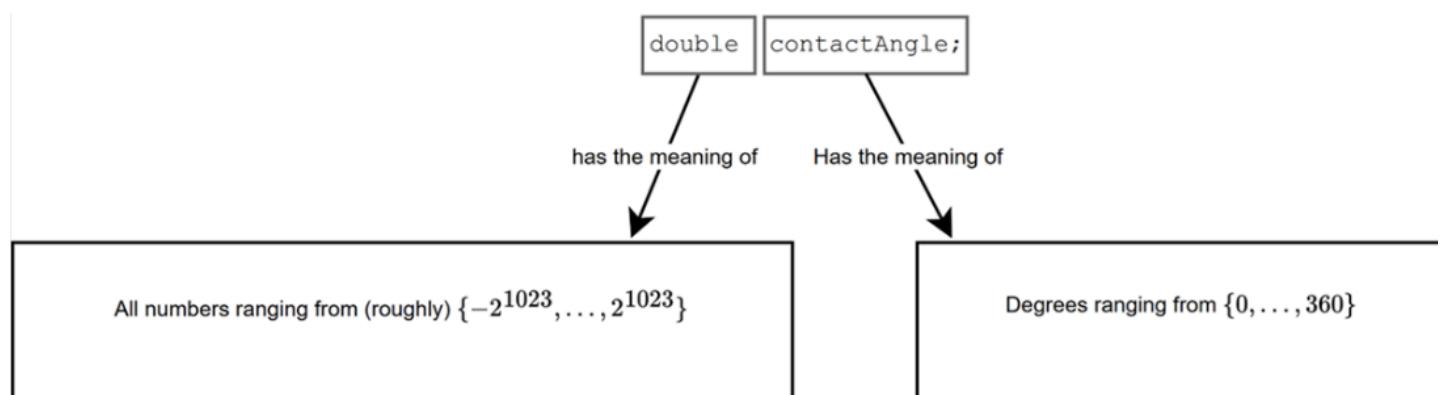


Figure 3.2 Conflicting meanings between our data and the type we've chosen to model it

If we hold up the meaning of our type (an absolutely massive set of positive and negative numbers) next to the meaning of our data (the tiny set of degrees between 0-360) the mismatch in their meanings is impossible to ignore. 99.9999%+ of all the values which are representable with a double are *completely invalid and wrong* for the domain idea we're trying to represent (degrees)!

This is one of the things that's extremely interesting and powerful about dealing with meanings. Despite not having written a single line of code, we've discovered a "bug" of sorts in our program, or perhaps more accurately, a place where bugs can live. The types betray our data's meaning and in doing so, cause our program to have more incorrect states it can enter than correct ones. These invalid states are the warm, damp breeding ground where bugs thrive.

To fix this, we have to align the meanings. We do this by introducing new types into our program. Let's fix the contact angle attribute by creating a type which enforces its constraints.

Listing 3.16 capturing our specific meaning of Degrees in a type

```
record Degrees(double value) { #A
    Degrees {
        if (!(Double.compare(value, 0.0) >= 0) #B
            && Double.compare(value, 360.0) < 0)) { #B
            throw new IllegalArgumentException("Invalid angle");
        }
    }
}
```

#A Capturing our domain information in a new data type

#B These are the same checks from listing 3.13, but now used to guard our domain specific type

Where do we use this type? Everywhere we want its meaning! Let's add it to our Measurement model and see what happens.

Listing 3.17 Encoding domain information about degrees into our model

```
record Measurement(
    String sampleId,
    int daysElapsed,
    Degrees contactAngle #A
){}
```

#A Swapping from double to Degrees and removing the redundant naming

Now we're really getting into something cool.

```
Degrees angle = measurement.contactAngle(); #A
```

#A The code now carries its own semantics around!

We no longer "forget" our data's meaning after we construct our Measurement type. In fact, we *can't* forget it, because the compiler won't let us! Its meaning now survives all the actions which previously caused it to become unclear or lost. We can freely program with our value without having to constantly re-validate it or defensively check that it's still what we expect it to be.

Listing 3.18 Types keep the meaning of our data in tact

```
Map<String, List<Degrees>> bySampleId = measurements.stream() #A
    .collect(groupingBy(Measurement::sampleId,
        mapping(Measurement::contactAngle, Collectors.toList())));

List<Degrees> totalChanges = bySampleId.values() #B
    .stream()
    .map(x -> minus(x.getLast(), x.getFirst()))
    .toList();
```

#A No more guesswork. The types are unambiguous

#B Even as we transform and reshape our data, the types keep the meaning explicit

There's another interesting thing here that we might notice. The code has started conveying more information about itself at the type level. Before this change, we had to read the implementation line-by-line to figure out how data moved through our system. Now we can skip all of that and skim "above" the implementation by just looking at the types. The more meaning you move "up" into the type system, the less important the implementation details of the code become. This reasoning "above the code" will

become a powerful tool in our arsenal as we build more sophisticated types. Eventually, we'll be designing entire chunks of our system without thinking about code at all!

Let's do the same thing with the Days Elapsed attribute. We know from section 3.1 what the data means, but does our type support that meaning? Let's compare.

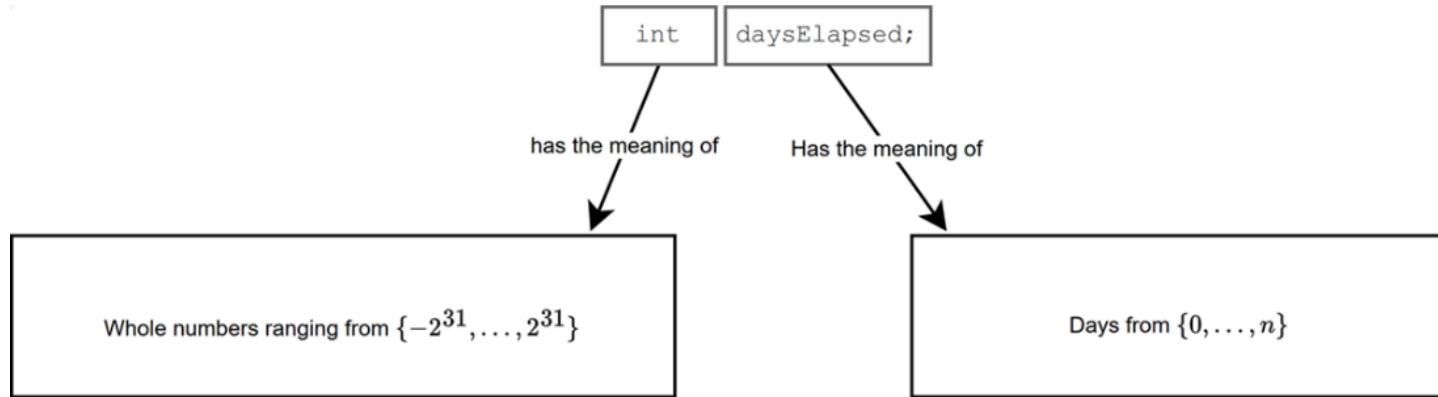


Figure 3.3 more conflicting meanings between our type and data

Not quite. In fact, we've got more-or-less the exact same problem as we saw with the contact angle field: most of the values our type supports are incorrect for our data. What we're trying to do is enumerate the number of days that have elapsed, but the int type, by nature of it being signed, allows negative values to be represented.

So, let's move this idea into the code by creating a new type that only allows non-negative integers.

Listing 3.19 Capturing the constraints in a new type

```
record NonNegativeInt(int value) { #A
    NonNegativeInt {
        if (value < 0) { #B
            throw new IllegalArgumentException(
                "Hey! No negatives allowed!"
            );
        }
    }
}
```

#A introducing a new record type to capture our meaning
#B And enforcing that only valid states can be constructed

And then we update the Measurement model with our new type information.

Listing 3.20 Putting the domain information into our model

```
record Measurement(
    String sampleId,
    NonNegativeInt daysElapsed, #A
    Degrees contactAngle
){}
```

#A Swapping int for NonNegativeInt

To my eye, this is starting to look really good. Even if those types did nothing to enforce their constraints behind the scenes, the code is still much, much richer for having them.

We're beginning to be able to tell at a glance what this code is supposed to mean. If we're dropped completely fresh into this code, we'd have a pretty good idea how to use it. The fact that those types *do* enforce their constraints is icing on the cake.

All that's left is encoding the meaning of the sample ID into our codebase.

3.3.1 Beware the String

The Sample ID field is an interesting one. As we do our "compare the meanings" exercise, if we try to pin down the concrete set of values that our String type denotes, we stumble into the fact that String doesn't really denote anything specific at all. It could be anything expressible in Unicode (an effectively infinite set of values!).

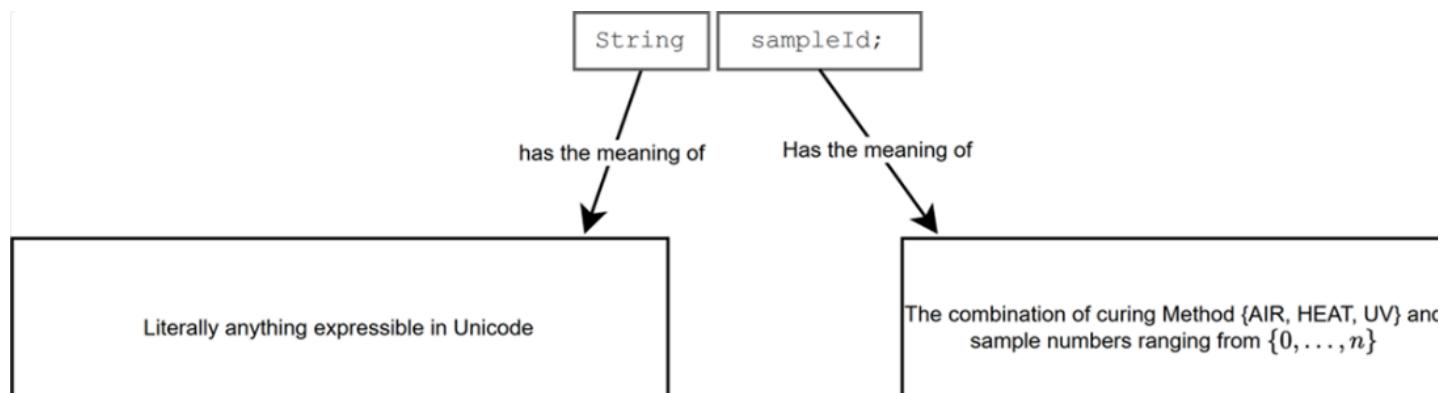


Figure 3.4 Sending very mixed signals

So, at this point, you know what we have to do: create a new type to encode the constraints! However, we've got to be cautious here. It's easy to slip into auto-pilot and start wrapping up everything in a new type, slapping on some validation, and calling it a day. The challenge we face as programmers is to make sure that we're not confusing the *mechanical process* of introducing types with the *intentional process* of giving the code meaning.

The approach we took in listing 3.13 was to validate the stringly form of the data during construction. If we pulled that same approach into a new type, it might look like this:

Listing 3.21 Capturing the meaning of SampleId in the code?

```
record SampleId(String value) {      #A
    SampleId {
        if (!sampleId.matches("(HEAT|AIR|UV)-\\d+")) {          #B
            throw new IllegalArgumentException(
                "Must honor the form {CuringMethod}-{number}" +
                "Where CuringMethod is one of (HEAT, AIR, UV), " +
                "and number is any positive integer"
            );
        }
    }
}
```

#A Introducing a new data type

#B Validating that the incoming string matches the shape we expect

Which we would, of course, dutifully stick on our Measurement model.

Listing 3.22 All done?

```
record Measurement(
    SampleID sampleId, #A
    NonNegativeInt daysElapsed,
    Degrees contactAngle
){}
```

#A Swapping the raw String for our new SampleId type

But are we done? Have we actually captured the meaning of this attribute in our code? I'd argue the answer is an unsatisfying "...kind of?". It's a step in the right direction, but it ultimately kicks the meaning can down the road. When it comes time to program with this new data type, we'll find that we still have that same "friction" we started with, it's just been moved to a different spot.

To give an example of this friction, let's try to do something very basic: bucket our data by its curing method. This would let us analyze how contact angles changed over time within each group, which would be pretty neat info to know. Should be pretty easy to do, right? Probably a one-liner with the streams API.

Listing 3.23 grouping by...?

```
measurements.stream()
    .collect(groupingBy(m -> m.sampleId().????)); #A
```

#A wait... we hit a dead end

Except we "lost" that information about the curing medium! Despite wrapping this up in a fancy new type, we're actually right back where we started: the code "forgets" what it is. The meaning only lives temporarily inside of the constructor while it's being validated. Once that process is done, what we're left with on the inside is "just" a String.

However, not all is grim. Our type *does* enforce that the string is in the right *shape*. The rest of our code can use that guarantee to reliably extract the data it needs. We can pull out the curing method while grouping by doing some basic string manipulation.

Listing 3.24 grouping by the internals

```
measurements.stream()
    .collect(groupingBy(
        m -> m.sampleId().value().split("-")[0] #A
    ));
```

#A Our type guarantees that the string is in the right shape, which allows us to confidently split on known values and do blind array access because we know it'll have values.

That works, but... it definitely doesn't feel right. We've got textbook information leakage going on. A completely unrelated part of the code is forced to know and care about the internal representation of our Sample ID. That's no good.

To work around that information leakage, we could move the extraction methods into the record's body.

Listing 3.25 Custom accessor methods

```
record SampleId(String value) {  
    #A  
  
    public String curingMethod() {  
        return this.value().split("-")[0]; #B  
    }  
  
    public String sampleNumber() {  
        return this.value().split("-")[1]; #C  
    }  
}
```

#A (constructor omitted for brevity)

#B Splitting our string to extract the target value

#C Same deal. This array access is safe because our validation ensures the string is in a known shape.

So far so good. Now our grouping method doesn't need to worry about any internal details.

```
measurements.stream()  
.collect(groupingBy(  
    m -> m.sampleId().curingMethod() #A  
));
```

#A Using our new accessor

But is this actually moving us in a better direction? Let's do the ol' gut check of just seeing what happens to our meaning when we assign something to a variable.

```
String method = measurement.sampleId().curingMethod(); #A
```

#A ugh... back to being a string. We "forget" our domain information

Ugh! All we've done with our changes in listing 3.25 is move the meaning back yet another level! What this data *means* is only defined at occasional and transient points throughout our code.

We could keep stamping these cases down one by one, but the fact that our code keeps forgetting its own meaning is a good smell that our problems are likely at the modeling level. We've fallen into a very, very common trap in programming: we've let how something happens to look while serialized completely dominate how we model it in our code.

Let's take a step back and remember what it means to be a sample ID:

Listing 3.26 Our finalized understanding of the data

CuringMethod:
One of: (AIR, UV, HEAT)

SampleID:
The pair: (CuringMethod, positive integer (0 inclusive))
Globally unique.

. . . #A

#A Other attributes omitted for brevity

This is the data we want in the type system. Nothing about what our data means implies String. The String is just an incidental artifact of how our data happens to be serialized.

```
{"sampleId": "AIR-1"} #A
```

#A We have to avoid getting anchored to how data looks, and instead focus on what it means!

Let's start over and build up our SampleId type from the data's *meaning*, rather than how the data *looks*.

We can go pieces by piece beginning with the curing medium. It's a fixed set of values, which leans really well to being modeled with an enum. Next, is the sample number, which is a positive integer. Luckily, we already made a type for expressing just this idea. So, we can reuse our NonNegativeInt type. Finally, we can construct our SampleId from these smaller pieces to fully capture the meaning of the data in the type system.

Listing 3.27 Capturing the meaning of Sample ID

```
record NonNegativeInt(int value){/*...*/} #A

enum CuringMethod {HEAT, AIR, UV}      #B
record SampleId(
    CuringMethod method,             #C
    NonNegativeInt sampleNum       #C
) {}
```

#A The NonNegativeInt type created previously

#B An enum captures the curing methods

#C We compose these types together to form our composite type of SampleId

And just like that, all of the friction in our code disappears! Our code keeps its meaning even as we transform it and assign it to different variables.

```
CuringMethod method = measurement.sampleId().method(); #A

measurements.stream()
    .collect(groupingBy(
        m -> m.sampleId().method() #B
    ));
```

#A No more ambiguous strings!

#B No more leaking implementation details

3.4 Is all this effort worth it?

Is it worth going through the trouble of creating these new types? I'll give the usual, very unsatisfying, and standard engineering answer: "It depends." I like to think of it as a Laffer curve. Too few types, and you have to solve a lot of avoidable self-inflicted problems. Too many types and, well... you also have to solve a lot of avoidable self-inflicted problems.

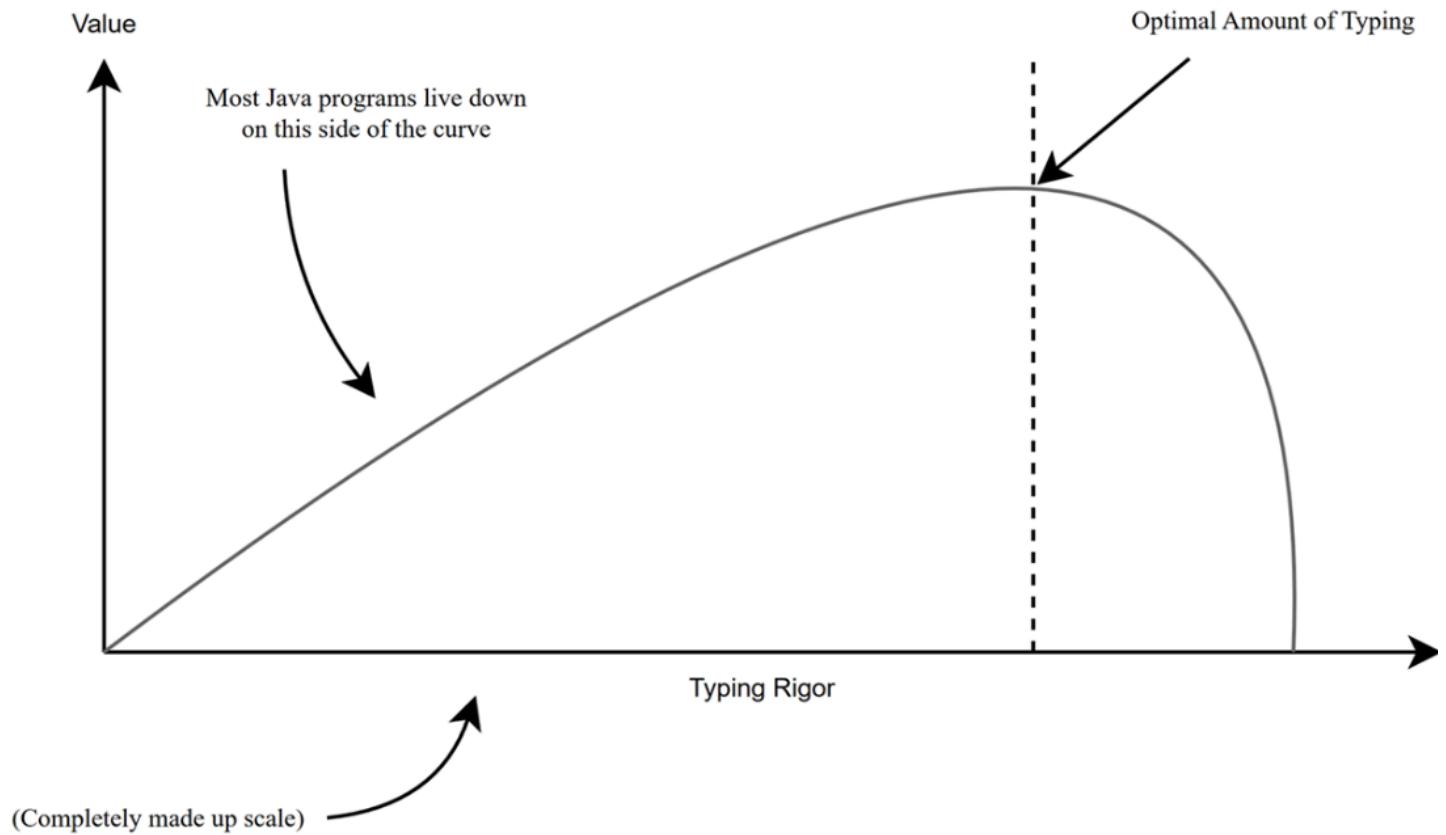


Figure 3.5 This curve is entirely made up, but how I generally think about things. “More types” = “more better” (up to a point)

Most Java programs are tragically under-typed and filled with dangerous gaps in their representation. These gaps cause painful and expensive errors that impact the real people that rely on our software systems. As such, it's a safe bet that additional types will more than pay for themselves in the beginning. However, there's a sweet spot that takes a bit of experience and fumbling around to find. You can definitely lean way too hard into the type safety world and start to drive yourself (and teammates) a bit mad. I've definitely gone down this route. It's important to remember that there will always be instances where you *don't* need a special type. It's A-OK for something to be a "plain" int or double. Sometimes an unconstrained String is just what fits the bill (though, I remain suspicious). All of that is perfectly compatible with the fact that there will be many more instances where introducing new types is the right call for the codebase.

The key thing is that we approach this as engineers carefully and thoughtfully managing tradeoffs. Many developers look at the upfront work required in defining new types and immediately jump ship to dismiss it under the flag of "boilerplate". They only look at the costs, while forgetting to weigh the value. This all-too-common tendency toward simple black and white thinking is unbecoming of our roles as engineers. Instead, we have to learn to take a step back and weigh the effect each of our modeling decisions will have on the code as a whole. Introducing a new type involves boilerplate for sure. That's undeniable (though records make it minimal). But it also creates value in terms of safety, correctness, and understandability. Those are the qualities that we weigh the boilerplate against.

3.4.1 Low hanging fruit?

You'll often find a common and pervasive resistance to additional typing based on the belief that, "at best," types can only catch the "lowest hanging fruit" in terms of bugs. This rationale is usually paired with justifications like "you need to write tests anyways" and (often stated smugly) "every bug that ever made it to production passed the type checker". These statements are, of course, true in that "technically correct" way- if you get a bug in production, then, by definition, it made it through the type checker and test suite. However, using these rationales as a justification for avoiding better typing reflects an over-correction. It can cause us to miss out on the special layer of additional defense and communication that types uniquely bring to the table. Even "low hanging" bugs can cause massive impact if missed.

One of my more blundering "low hanging misses" caused a system I worked on to start emailing scary messages to AWS customers saying that we'd charge them fees up to 150% of their annual bill's total.

What made it such an avoidable blunder is that everything about the code was completely correct. There was no logical error. The "bug" came from outside of the code in the form of configuration. What a particular variable meant was misunderstood by one of the developers, and as a result was constructed with a value which was valid for the data type, but semantically invalid for what it was actually supposed to mean. Can you see the bug in this code?

Listing 3.28 Is this code correct?

```
double computeFee(double total, double feePercent) { #A
    return feeAmount = total * feePercent; #B
}
```

#A Will this always do the right thing?
#B What kind of tests would you write?

To operate correctly, the code in listing 3.28 depends entirely on everyone's understanding of what those doubles are supposed to be. What happened in our case is that the person who wrote the code relied on the "obvious" way of representing a percentage. Because it was "obvious", the meaning of those doubles was never put into the code. It only lived in the head of the original developer. So, that meaning was lost when the code changed hands. Another team member had an alternative mental model for the "obvious" way to store percentages, and thus, what one developer expected to be 0.015, became 1.5, which eventually became formatted into an email as "150%", resulting in a bunch of confused and unhappy end users.

Now, you have two paths you could take for assigning "blame". The first is on the humans managing the code. They should have simply been "better" programmers. They should have thought more about their inputs and considered edge cases. Or they should have written more tests -- or written better tests. Or they should have read the docs on the objects, or done whatever action hind sight tells us that a "good developer" should have done to prevent this "low-hanging" bug. However, all of these "should haves" reduce a complex system down to a single point of failure: the one human who made a perfectly valid choice given the code in front of them. Telling them not to step on the mines we buried in the ground doesn't do anything to move the needle forward or fix the root problem.

The other option is to view the problem as being that the code didn't tell anyone what it needed in order to behave correctly. Instead, it was left as ambiguous guesswork for the humans to tackle. If we imagine a world where what the original developer meant by "percent" was captured as a type, would we have still faced that same problem in production?

Listing 3.29 A better world

```
record Percent(double numerator, double denominator) { #A
    Percent {
        if (numerator > denominator) { #B
            throw new IllegalArgumentException(
                "Percentages are 0..1 and must be expressed " +
                "as a proper fraction. e.g. 1/100");
        }
    }
}
```

#A Being exceedingly explicit with what the type represents

#B Our constraints would have immediately caught the error.

Of course not! In this world, the problem never happens -- because the possibility for the problem has been completely removed. The code forces us to deal with the fact that there is a special expectation here. We can't just plug in 1.5 or 150 or however we think a percentage should be represented, the type signature doesn't allow us. We're forced to express our percentage as something that can only evaluate to a value between 0 and 1. Anything else and it pops an error informing us of our mistake.

3.5 What about performance?

Some developers might recoil in abstract horror at the thought of all of these additional object types. The fear is that the extra object allocations will make their programs "slow". This mind set can be a dangerous one. Knuth's "premature optimization is the root of all evil" is one of the few near-universal truths in software development. The "evil" it causes here is that we'd silently reject more precise modeling because of unqualified anxiety about performance. We're letting unstated fears do our design, the result being less reliable and maintainable programs. If we're going to trade reliability for performance, we should do it with full understanding of the costs and benefits.

If you're still in doubt, try it! Don't let vague worries or superstition cloud your design process. Create a few million objects and stuff them into a list. Then transform those objects into some other objects. Then do it all over again a few thousand times while recording the durations. If you look at the numbers after doing all of that, I bet it'll be hard to end up with any opinion other than: "the JVM is crazy good at this stuff". The performance "tax" of allocating a few million Doubles versus a few million "complex" data types like SampleId is all of a few single digit milliseconds even on my ancient and underpowered laptop. So, let's focus on making the code correct. We'll let the JVM worry about making it fast.

3.6 Wrapping Up

In this chapter, we explored how the pursuit of meaning is fundamental to the software development process. The meaning is what we use when modeling a domain. If we

know what something is, then we know what it must not be. Meaning leads to constraints, constraints lead to invariants, invariants lead to code.

We also learned that this lens of meaning can be turned toward the type system itself. This gives us an extremely simple and powerful way of checking that our modeling is sound. If the meaning of our type lines up with the meaning of our data, we're usually in pretty good shape. If they conflict with each other, it's usually a pretty good smell that our modeling is off.

Next up, we're going to start to explore how we can use what something means to push our modeling into even more powerful territories. So far, everything we've done has been about creating types which do most of their "work" at runtime via validation in their constructors. We're going to learn how we can move what something *is* even earlier into the development lifecycle: we'll learn how to make illegal states impossible to represent.

3.7 Summary

- The most important part of our job is getting to the bottom of what data means
- We have to go outside of our programs to find data's meaning. Inside we can only model it
- Understanding what the data in a domain means is fundamental to writing good software
- Simple questions, applied thoughtfully, can help us explore a complex domain
- When we put data's meaning into code, it can prevent us from making errors
- Documentation, comments, and variable naming are useful, but they cannot enforce correctness or meaning on their own
- Data types are where we encode the semantics of what our data means.
- Putting validation code in the constructor ensures only correct data can be created
- Types have meaning just like data.
- We explore the meaning of a type by visualizing the concrete set of values it denotes
- Comparing meanings is a powerful tool for checking if our modeling is sound
- There's a sweet spot for typing rigor. Too many types is obnoxious, too few is dangerous
- Good code should make doing the wrong thing very hard (or impossible) to do
- Code which relies on all developers to share the same unstated opinion about the "obvious" meaning of an ambiguous type should be refactored to make the "obvious" explicit
- The JVM is extremely efficient. The performance impact of adding more types is seldom an issue
- Always pay caution to strings. They make it easy to accidentally hide aspects of a domain

4 Representation is the Essence of Programming

This chapter covers

- The walk through the messy design process
- The effect of AND on our designs
- Modeling options with OR
- Making Illegal states impossible to represent

Rather than a big bang “here’s what data-oriented design is,” we’re going to use this chapter to walk through designing a small feature in all of its chaotic glory. Data oriented design is quite a bit like all design processes: iterative and messy. We’ll make plenty of missteps, hit lots of dead ends, and have to do more than a few total resets. We’ll walk through all those messy bits here because, well, firstly, we’d do them in real life, too (at least when I’m the one doing design). However, beyond that, we walk through all the “wrong” parts so that we can train ourselves to become sensitive to what “wrong” feels like and the symptoms it causes in the code. Once we can put a finger on why something is wrong, making it right is much easier.

What might be kind of surprising is that our design process will be focused entirely on the data in our domain and what it means. Data Oriented design is largely in the spirit of Fred Brooks’ classic observation from Mythical Man Month: “representation is the essence of programming.” If we get the representation of our data right, everything else tends to fall into place. As such, we won’t worry about behaviors, or efficiency, design patterns, or any other operational focused concerns for now.

We're also going to ignore that our job as software engineers is to build *systems*.

Systems force us to juggle complex, often unsatisfying design tradeoffs. We'll deal with that mess later. For now, we're going to allow ourselves to design in glorious isolation of the world. We'll focus on just the data, what it means, and see how much we can accomplish by tweaking its representation.

4.1 Starting from Requirements

To give a motivating example, I did a brief stint working on manufacturing software for a rocket startup. One thing that's interesting about rocketry is that beneath all of the advanced CAD software and simulations, the core technology that underpins absolutely everything is the humble and very low-tech checklist (technically, they called it a “procedure”, but I know a checklist when I see one). Almost nothing happens without someone somewhere marking off a checkbox as they go. Rockets are just too complex for humans to manage without them. The checklist's reign is absolute because, as you might guess with rockets, a lot of those individual steps are really important. You don't want questions like “did we remember to...?” when the “remember” part involves things that explode.

So, we're going to walk through modeling one of these checklists that keep rocket manufacturing on track. However, we'll do so with a slight twist: unlike our previous chapter where we started with some existing code and refined from there, we're going to start completely from scratch with nothing more than a loose requirements statement. Extra heavy emphasis is placed on the "loose" part, as that's unfortunately the shape in which most requirements will end up in front of us. Our job is to tease apart everything that's vague and under-defined so that we can turn it into precise code.

So, let's get started. Here's our "official" requirements statement: "I want to be able to create and use checklists."

4.1.1 Revisiting the Humble Checklist

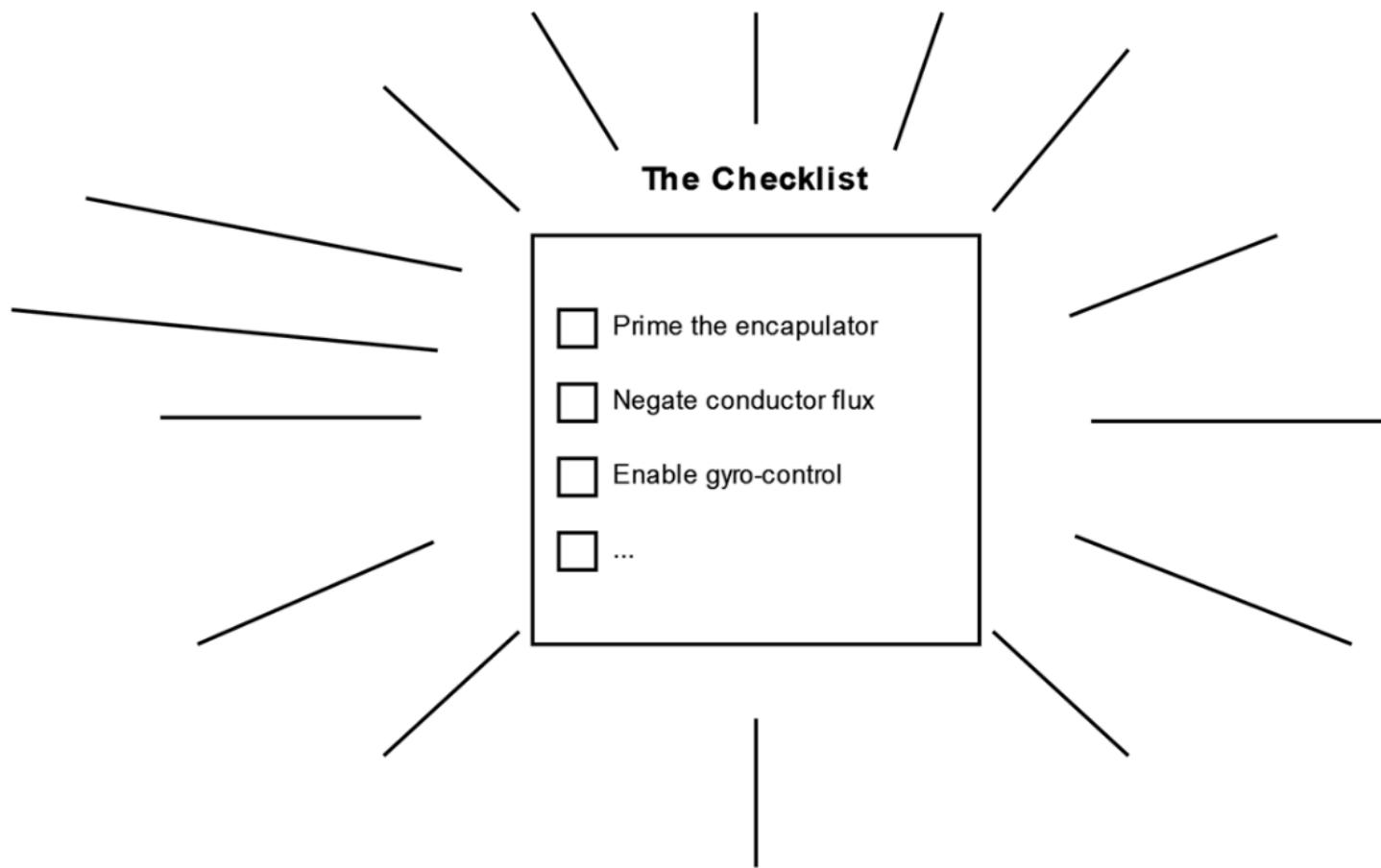


Figure 4.1 The checklist!

What's a checklist? Well, it's a list! And it has checkboxes! With labels! And we check them off as we complete stuff!

It's so well-trodden that, if we wanted to, we could probably dive straight in and sketch out our view of a checklist with some code like this:

Listing 4.1 Done!

```
record Step(String name, boolean isComplete){} #A
```

```
record Checklist(List<Step> steps){} #B
```

**#A A step is a named task that's either not started or completed
#B And a checklist is just a collection of individual steps. Easy!**

However, if we present this grand design to our friendly rocket scientists, they'd let us know that it doesn't line up with how they *use* checklists. The reason they use checklists is repeatability. They want to figure out the hard stuff once, and then reuse that knowledge over and over. So, there are two distinct phases: the first is coming up with a template for the work, and the second is having people run through instances of that template to perform the work.

Checklists have enough going on that, to quote Leslie Lamport, "it's probably a good idea to know what we're doing before we try to build anything." So, let's pause on the implementation and try to figure out what we're going to be building.

4.1.2 Firming up loose requirements

We're still sort of fumbling around at this point (which is normal), so we'll expect a few missteps, but let's charge forward and write down what we know. We've learned that there are two distinct things going on here, so we can split out the idea of a template from the idea of an *instance* of the template.

Listing 4.2 Writing down what we know

Template:

A collection of things to do

Instance:

A run-through of a template where
we complete the things to do.

Those are the broad strokes. Now it's just a matter of making incremental steps towards a deeper understanding of what each one means. As usual, we can just pick something and start asking questions. Starting at the top with the templates, we can meet with our rocket people and ask how they actually use these things. We learn that they've got templates for just about everything. If it needs to be repeatable, there's a checklist for it. It could be as low-level as how much to torque a particular nut in some sub-assembly, to as high level as transferring flight control to the rocket prior to launch. All of which tells us that we need a way of uniquely keeping track of these templates.

Listing 4.3 Refining the understanding of Templates

Template:

A named collection of things to do #A

Instance:

A run-through of a template where
we complete the things to do.

#A Qualifying that our templates will be identified by their name

This description of templates is a good enough start for our purposes. We next prod the rocket folks to learn how they use instances of these checklists in practice.

This turns out to be pretty revealing, because they use them slightly differently from the informal way you and I might use a checklist. For instance, if I'm preparing for an international flight, I might run through a checklist as I pack (socks, passport, toothpaste, etc.). But once I'm done, I'm done. In the trash it goes. However, this is very much not how the rocket people use checklists. They run their checklists over and over again -- often directly back-to-back as they drill operations. Each time they run through a checklist, they don't throw it away (like I do) or just clear it out and start over (like I would), they keep them as *data*. "On this date, for this test, we performed these actions." Once completed, they're important immutable artifacts cataloguing which operations were performed as part of a particular test.

Listing 4.4 More refining of what instances of our checklist are

Template:

A named collection of things to do

Instance:

A **named** run-through of a template #A
at a particular **point in time** where #A
we complete the things to do and
record the results.

#A Tacking on the two important aspects of how checklists get used in our particular domain. We have to be able to track when things happened and why

Putting it all together, we have:

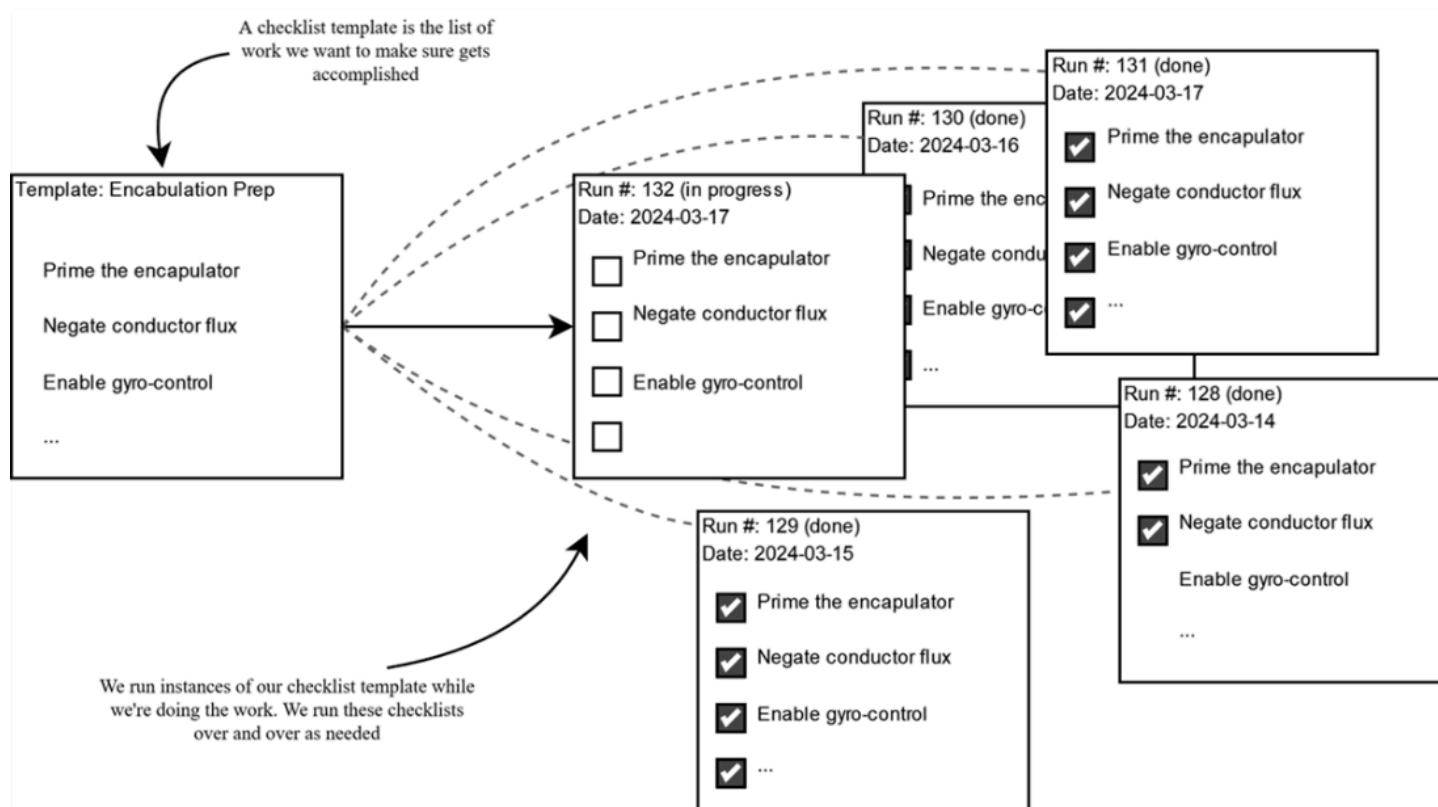


Figure 4.2 A more complete view of checklists

4.1.3 Translating into a model

Now that we have a pretty good high-level idea of how checklists get used within our domain, we have to decide how we'll model it. This is a tricky part of the process. From our high-level understanding, which is described by just those few loose sentences in listing 4.4, we have to abstract out a set of attributes which, when combined together, capture the core essence of what it means to be a checklist.

In the previous chapter, we approached modeling in two distinct phases. First, we wrote down what we knew about the meaning of our data. Then, once we were sure we it was clear, we swapped over to writing code. This let us use the meaning of our data to guide the design of our code.

In this chapter, we'll go with a less strict approach that freely moves back and forth between the two phases. This is much more representative of how I would actually approach modeling a new domain. We'll sketch out a few records, and then take a step back and look at it from the "meaning" perspective, then swap back to code to sketch some more, take another step back, move things around, etc. etc.

The main point I want to stress here is not to think of this process as actually "*writing*" code yet. We're using code purely as canvas for our thinking. Right now, it's wet clay. It's malleable and coarse. We definitely won't build anything on top of it yet. We're only sketching. We'll add, remove, and shuffle it around as we explore how each change affects the meaning our data conveys.

So, let's take a stab at lifting out a set of attributes from our current understanding that we can use to model these domain ideas.

Listing 4.5 Figuring out what's important

Template:

A **named** collection of things to do #A

Instance:

A **named** run-through of a template #A
at a particular **point** in time where #A
we **complete** the things to do and #A
record the results.

#A Not terribly different from what we might do in Object Oriented design. we can look through and pick out some noun-y words / ideas that pop out as key to the domain

We can start anywhere, but let's first tackle the Templates. They model collections of "things to do". We already took a stab at modeling these "things to do" as a record called **Step** back in listing 4.1, so we could plug that in as a starting point. We might not keep it, but that's ok. Throwing things out is part of the process.

For the templates themselves, they're just collections of these steps grouped under an identifiable name, so we might sketch out all of it like this:

Listing 4.6 Attempt #1: Sketching out the Templates and steps

```
record Step(          #A
    String name,      #A
    boolean isCompleted #A
){}

record Template(      #B
    String name,      #B
    List<Step> steps  #B
){}
```

**#A Reusing the model from our initial stab at this in listing 4.1 just to kick things off
#B Steps live on a Template, rather than what we ambiguously called just a "Checklist" before.**

This is a pretty good start, but it's not where we *stop*. (I'll keep belaboring the point: we're still in the exploring / design phase. We're using code as a thinking tool). We've written a little code, so now we take a step back and see if that code is capturing the meaning of our data.

If we review our definitions from listing 4.4, we've said that a template is "a named collection of things to do". It only cares about what the steps *are*. It doesn't say anything about how they're used or completed over time. However, our code says something different. We've got this `IsCompleted` idea hanging around in there. It's superficially small, but this coupling allows our code to "say" rather weird things when we're constructing templates.

Listing 4.7 all socks are already always packed

```
Template travelPrep = new Template(
    "International Travel Checklist",
    List.of(
        new Step("Passport", false),    #A
        new Step("Toothbrush", false),   #A
        new Step("bring socks", true),  #B
    )
);
```

**#A I could kind of buy this being here as long as they were all reliably set to false, but...
#B What the heck would it mean if any are set to true? This step is already always completed? All socks are already pre-packed for all trips?**

If we just focus on the *code*, we could probably get away with this modeling even though it doesn't quite line up with what we've defined a Template to mean. However, we have to be extra cautious of falling into this trap! We always have to use the meaning of the data to guide us through our design phase. Without it, pesky "code" thoughts can start to invade our thinking.

These "code" thoughts usually start with something innocent sounding like "well, we *could* just...", but where the "just" part is usually defensive programming in *other* parts of your code to work around the problems being introduced in *this* part of the code. For instance, upon seeing the strange "socks are already always packed" problem, it could be justified as "well, we *could* always just ignore what's in the template and...".

Listing 4.8 We could always just defensively make sure that...

```
new Instance(template.stream()
    .map(step -> new Step(step.name(), false)) #A
    .toList());
```

#A We haven't modeled our checklist Instances yet, but we could imagine making sure that creating one involves defensively rewriting any pre-completed steps in our template

But let's not fall into that trap! We want our code to enforce its own meaning and help us, the fallible programmers, not make mistakes. So, rather than patch around this modeling problem with defensive code, let's *remove the problem entirely* by just using better modeling. Of course, we don't quite know what that better modeling looks like yet, but we do know what's wrong with *this* one: it allows bad states. So, let's take `IsCompleted` out of there.

Listing 4.9 Cleaning up the template model

```
record Step(
    String name
    #A
) {}
```

#A Removing IsCompleted. We're not totally sure where it goes yet, but we do know it doesn't go here.

Now we have to figure out where to put it. If we flip back to our definitions in listing 4.4, it's the *instances* of our checklists where we record the individual steps as completed, not the templates. So, maybe if we tackle sketching out a model for `Instance`, we'll find where `IsCompleted` fits.

Listing 4.10 Another attempt: modeling the checklist Instance

```
record Instance(
    String name,          #A
    Instant date,         #A
    Template template    #A
    #B
) {}
```

#A One of the many ways we could model the Instance

#B Do we store something like IsCompleted here somehow?

The model of the `Instance` itself feels OK, but it's still not immediately clear where something like `IsCompleted` would fit. It's tricky. It definitely *feels* like a `Step` and its status (`IsCompleted`) belong together, but we know that weird things happen if we attach the status to the `Step` directly like we did back in listing 4.6 (our templates start to "say" nonsensical things). So, we might try going the other way: introducing a new type which can hold a `Step`.

Listing 4.11 Trying a new approach: using a new type to track the status of a step

```
record Status(
    Step step,           #A
    Boolean isCompleted #A
) {}
```

#A Now a Step and its status are back together again, but without the problems that our original modeling caused

That feels pretty good, but we still haven't tied it explicitly to our Instance. But... maybe we don't need to! Every model we create is an approximation of the real world. It needs to capture its meaning, but it doesn't need to match its shape. So, what if we just left Statuses as their own model and referenced them elsewhere as needed?

One argument for this approach is that these status records will be the main thing that gets continuously updated as our program is run and people work on their checklists. All other data (once it's created) is completely static. So, we might embrace that fact and let statuses be their own thing which can be stored and tracked in whatever form makes the most sense for our program.

```
class MyCoolChecklistInMemoryStorage {  
    private Map<Step, Status> statuses; #A  
}
```

#A In-memory storage for tracking our checklist app (or something like this)

We've written some more code, so here's where we flip back to thinking about our data's meaning again. It is the meaning that we want to drive the design, not just what "feels right" with code. So, let's ask something fundamental: what does it mean to be a Step in a checklist? For instance, do they need to be unique within a template? What about across different templates? A checklist for international travel and a checklist for a local weekend getaway might both have a step called "pack socks". However, you probably wouldn't have that same "pack socks" multiple times within the *same* template. This meaning of step tells us about some important constraints we need to enforce, but it also gives us feedback on how we've designed our Status type. We didn't model it with enough information. In order to correctly tie a Status to a *particular* step, we also need to know the template to which it belongs.

Listing 4.12 Adding template to our Status model

```
record Status(  
    Template template,      #A  
    Step step,  
    boolean isCompleted  
){}{}
```

#A Adding a reference to Template so we can tell which Step we're talking about

Listing 4.13 The finished data model

```
record Step(String name){}

record Template(String name, List<Step> steps){}

record Instance(
    String name,
    Instant date,
    Template template
){}

record Status(
    Template template,      #A
    Step step,
    boolean isCompleted
){}
```

#A Adding a reference to Template so we can tell which Step we're talking about

WHAT ABOUT THE REST?

To keep things brief(ish), I've purposefully left all of the constraints out of the example code. This means that our types are very similar to those we started with in Chapter 2: loose and dependent on implicit assumptions in order to be correct. For instance, all those strings are suspect (as usual!). Should they be completely unbounded? Would 12 fire emojis be a valid template name in our domain? (It'd be cool if so). What about making sure our steps are unique within a list? That's a pretty fundamental requirement for what it means to correctly construct a template!

I leave all of those constraints as an exercise to you, the reader. You should take a stab at turning the loose sketch we made here into code which knows how to guard its own invariants. Along the way, you might find that certain modeling decisions we made here force us to do some defensive validation during construction that itself might be eliminated by alternative modeling. We didn't go down some of these paths here because they rely on more advanced techniques which we'll explore later in the book. However, learning to see where defensive code can be swapped for better modeling is one of the most valuable skills we can develop as programmers, so the more we practice, the better.

4.1.4 New Requirements: Keeping Track of Who Did What

Our Checklist software is a huge hit. As more and more people use it, we learn something new about our rocketry domain: their checklists are *massive*. For instance, running something like a wet dress rehearsal, where they load propellant into the rocket and run a full countdown sequence, involves a dizzying array of individual steps which all have to be coordinated across multiple teams.

So, there's a new feature request. They need to be able to keep track of who completed which step. There should be some basic auditing of which user clicked the button and when.

Listing 4.14 Updating our requirements

Instance:

```
A named run-through of a template
at a particular point in time where
users coordinate to complete the list
of things to do and
record the results.

The user and time of completion must be recorded #A
```

#A New requirement!

If we're in straight programmer mode, this sounds like a trivial task – borderline mindless, even. Just another attribute or two. Of course, we will make sure we understood what these new additions *mean* so that we can accurately model them, but, after that, well, it's two attributes. How hard could it be? Just add them onto the model?

Listing 4.15 An easy addition...?

```
record User(String value){} #A

record Status(
    Template template,
    Step step,
    boolean isCompleted,
    User completedBy      #B
    Instant completedOn   #B
) {}
```

#A For sake of brevity, this will be our “good enough” User model.

#B Adding the new fields

Is it really that straight forward, though? We might not be able to articulate it just yet, but *something* about these new attributes feels different than the other attributes. As their naming implies, they're only relevant once the procedure has been completed, and yet they're always present on our model. So, it raises an interesting question: what do we put in there before a Step is completed?

```
new Status(
    template,
    step,
    false,
    ??? #A
    ??? #A
);
```

#A What do we put here?

To most Java developers, this might seem a bizarre non-question. It's *obvious*. We use nulls for data that's not available yet. However, this is, for lack of a better term, “code first” thinking. Java is the medium, but we're using it to model *data*, and that data has meaning that we need to capture. We have to train ourselves to notice when the code we're writing causes the meaning of our data to get lost. This is what happened when we introduced those two nullable fields. There was a subtle shift in the data's meaning. It's now *contextual*. If certain fields are set, then it means one thing. If other fields are set, then it means something else.

This blurring of our data's meaning isn't just being pedantic, it has caused us to take a substantial step backwards. We're trying to use our modeling to get *rid* of invalid states and unnecessary defensive programming, but the introduction of these fields has done the exact opposite: it has *introduced* new invalid states to our program! We now have fields which can be set when they shouldn't be, or *not* set when they should be. This is now more stuff we have to defend against during construction.

Listing 4.16 breaking things

```
new Status(  
    template,  
    step,  
    false,          #A  
    new User("Bob"), #A  
    Instant.now()   #A  
);  
  
new Status(  
    template,  
    step,  
    true,           #B  
    null,           #B  
    null            #B  
);
```

#A We've broken causality and somehow added a "completed by" user before the step was completed.
#B And then the opposite problem: the step is completed, but who did it is nowhere to be found

Code like this is frustrating because it forces us to not trust our own eyes. Records are supposed to be plain what-you-see-is-what-you-get carriers of immutable data. This is what makes them so easy to reason about. However, this modeling has caused some of those "easy to reason about" qualities to be lost. There's a disagreement between what the code *says* makes up a Status, versus what *actually* makes up a Status. There are secret rules in play (i.e. only some of the fields on this model are used some of this time). It's left to us to figure out *when*.

It's not ideal, but it also isn't the end of the world. We've only created two additional potentially incorrect branches we need to cover. And two is a pretty small number. We can use our usual tools to defend against them at construction time.

Listing 4.17 Guarding against missing (or not missing!) attributes during construction

```
record Status(  
    Template template,  
    Step step,  
    boolean isCompleted,  
    User completedBy,  
    Instant completedOn  
) {  
    Status {  
        if (isCompleted && (completedBy == null || completedOn == null)) {  
            throw new IllegalArgumentException( #A  
                "completedBy and completedOn cannot be null " + #A  
                "when isCompleted is true" #A  
            );  
        }  
        if (!isCompleted && (completedBy != null || completedOn != null) ) {  
            throw new IllegalArgumentException( #B  
                "completedBy and completedOn cannot be populated " + #B  
                "when isCompleted is false" #B  
            );  
        }  
    }  
}
```

#A Making sure things are set when they should be

#B and not set when they shouldn't be

Alright, now we're sort of back in business. The code once again keeps us on track. If we accidentally forgot to supply a user, the code will let us know.

Listing 4.18 No more ill-defined data

```
new Status(  
    template,  
    step,  
    null, #A  
    Instant.now()  
) ;
```

#A Now explodes with an error if we accidentally forget to supply a user

However, it's still kind of unsatisfying. We've lost *something* about our data's meaning with our modeling. A crack has shown in our foundation. It's not a big crack, but it's a crack nonetheless. One that we had to patch with that general purpose mortar that is defensive coding. But there are sprint deadlines, so we move on.

4.1.5 Increasing complexity: another requirements change

Time passes and we receive a new requirement. The people who manage these workflows need the ability to skip individual steps. Given the stakes involved, skipping a step doesn't come lightly. In addition to recording who did the skipping and when, we also need to store an auditable trail of *why* it was skipped.

This again sounds pretty easy. We've already got users and timestamps on the model for handling when it gets completed, so this is just more of the same – some copy/paste. The only new thing here is the rationale for why it was skipped, which we'll

have to talk to some of our rocket people to understand what they're after, but we could go ahead and drop those new fields into the model now.

Listing 4.19 Another easy update?

```
record Status(  
    Template template,  
    Step step,  
    boolean isCompleted,  
    User completedBy,  
    Instant completedOn,  
    Boolean isSkipped, #A  
    User skippedBy,    #A  
    Instant skippedOn, #A  
    String rationale   #B  
) {}
```

#A The basic metadata for who did the skipping and when

#B and lastly, the rationale for why the step was skipped

4.1.6 Refactoring the code, rather than the meaning

Your Don't Repeat Yourself (DRY) sensibilities might be twitching from the code in listing 4.19. Things that are *almost* the same causes a visceral reaction in the mind of the developer. We have to – no, we *must* -- get rid of duplication in all of its forms (y'know, *or else...*). So, what we might do at this point is look at what we've written down and start playing around with factoring out the "duplication".

The first thing we might do is factor out the similar attributes.

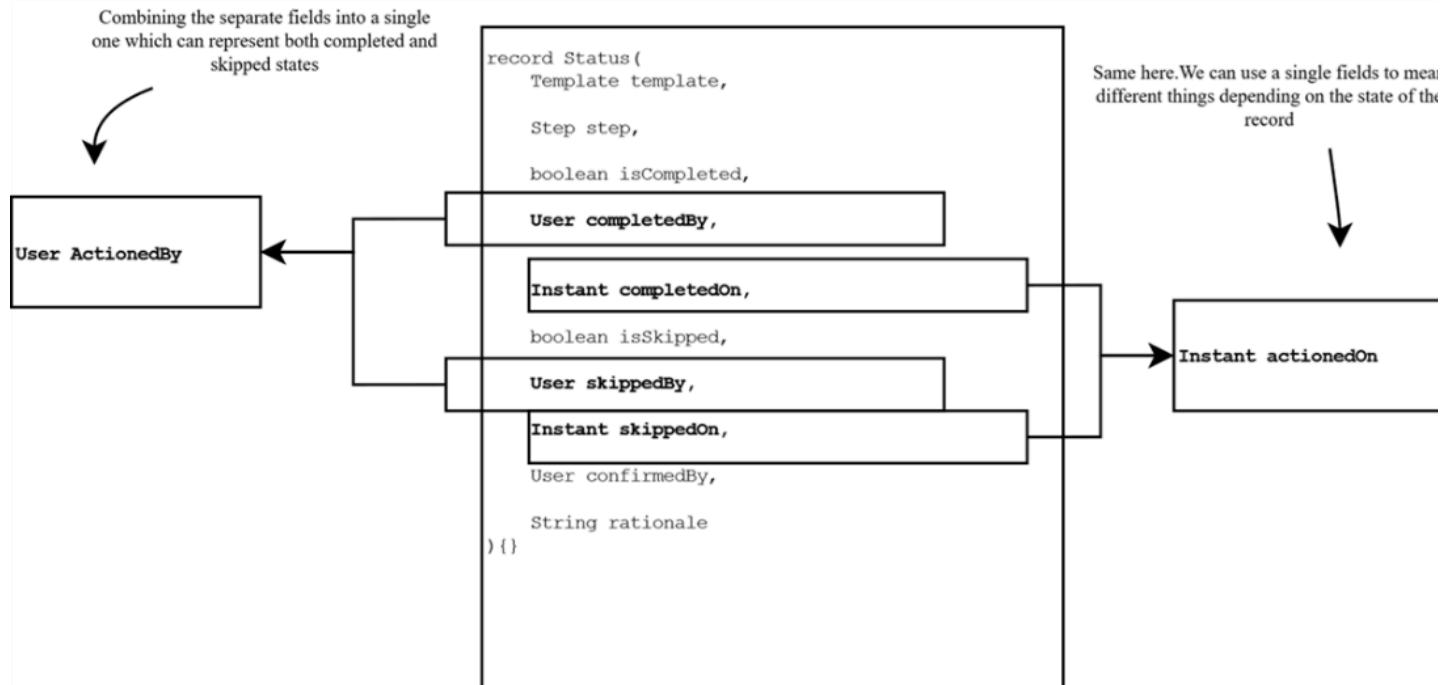


Figure 4.3 refactoring the attributes

And do the same thing with the two Booleans. we can collapse them down into a single enum.

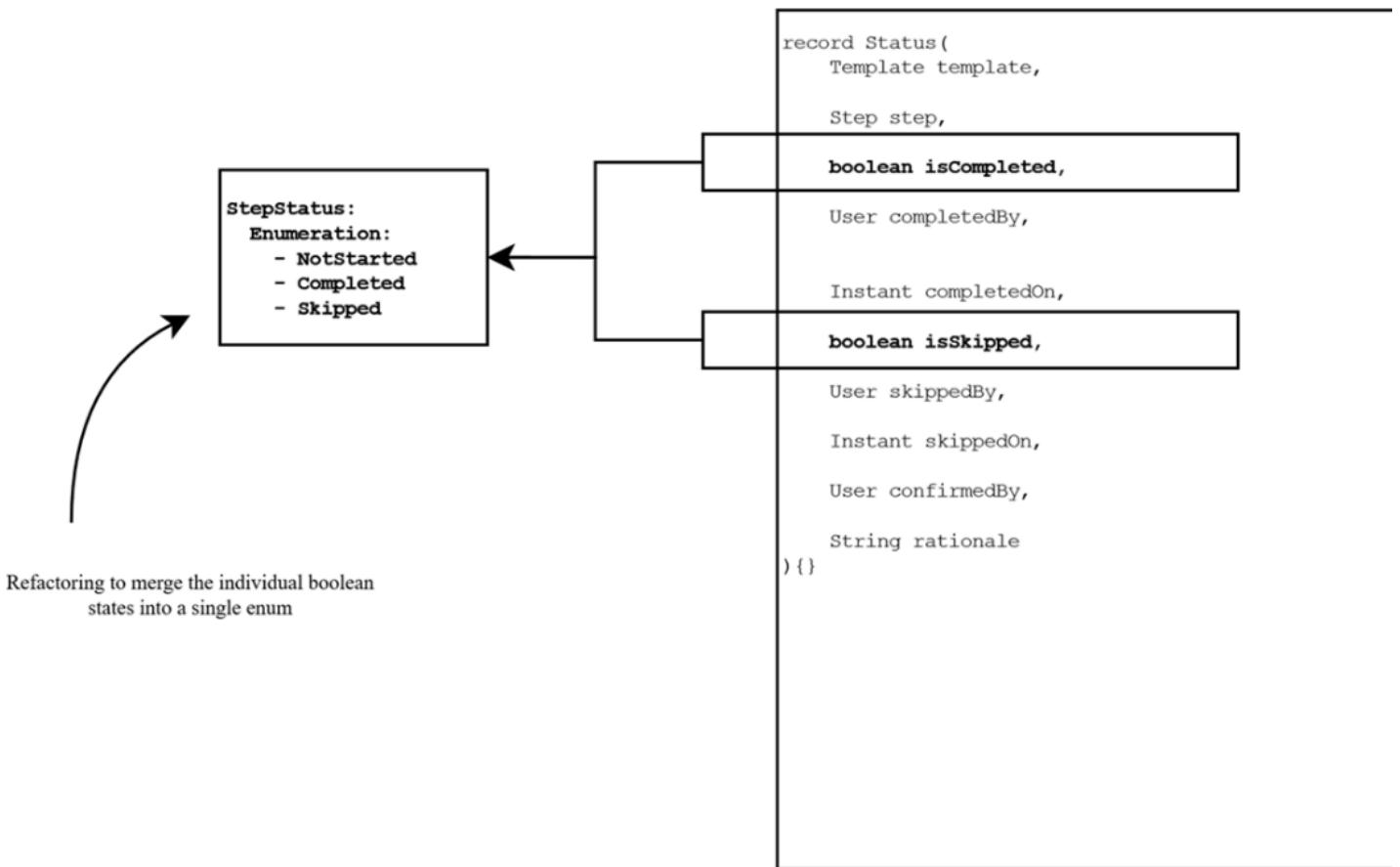


Figure 4.4 refactoring to remove the individual Booleans

The changes in figures 4.3 and 4.4 get rid of some of the more egregious “duplication.” We still have a bunch of contextual and obnoxiously nullable fields, but at least there are fewer of them than there otherwise would be.

Listing 4.20 Attempting to “solve” the redundant fields by DRYing the code

```

enum State {NOT_STARTED, COMPLETED, SKIPPED}; #A
record Status(
    Name name,
    State status, #A
    User actionedBy, #B
    Instant actionedOn,
    User confirmedBy,
    Rationale rational
) {
    /*...*/
}

```

#A Replacing the Booleans with an enum

#B Combining the individual completed on/by fields into a one

Without question, it’s much more DRY. There’s less code, which is nice. But... has it actually solved the root problem? I think we’d have to say no. The same set of problems that were plaguing us with the last round of attributes we added remains: the meaning is contextual. Our refactoring has slowed the rate at which things get worse. But they’re *still getting worse*. Despite DRYing things, we’re still contending with illegal states that only exist because of how we modeled the data. We still need to defend against them in the constructor.

Listing 4.21 Protecting ourselves from the illegal states our modeling allows

```
record Procedure(*/*) {
    Procedure { #A
        if (state.NOT_STARTED) { #B
            // ...
        }
        if (state.COMPLETED) { #C
            // ...
        }
        if (state.SKIPPED) { #D
            // ...
        }
    }
}
```

#A Ok... Here we go.

#B First we have to make sure that if we're not started, then everything else is null (because we shouldn't have any data yet)

#C If it's completed, we have to make sure the fields we want are not null, but also make sure the fields we don't want are null.

#D Same as C, but reversed. Make sure nulls are only where we want them, and not where we don't want them

If we need to add more statuses in the future, say, for tracking which procedures are Started or Blocked, the amount of validation we have to do (and our chance of getting it wrong) is going to just keep growing. We'll soon be drowning in `if` statements.

Further, the construction validation woes are just part of the problem. The bigger tax is the one we pay everywhere else in our codebase. We've fallen back into the problem we were trying to solve in the previous chapter where our code "forgets" what the data type means as soon as its constructed.

Anywhere we want to use our data type, we have to defensively check (or rely on human reasoning powers) in order to avoid making a mistake. This defensive coding will spread like a virus to each and every place we interact with our data. If we don't defensively check to enforce our assumptions about the data, we leave a minefield of potential null pointer exceptions for others to stumble on as they try to dereference things which aren't actually present on the model.

Listing 4.22 Ongoing woes

```
void doSomethingWithCompleted(Status status) {
    if (!status.equals(Status.COMPLETED)) { #A
        // now we know we can safely read actionPerformed
        // without a Null Pointer getting thrown
    }
    throw new IllegalArgumentException("Expected completed"); #A
}
```

#A How all our code will end up looking if we want to avoid accessing fields that aren't actually populated. Or! It won't look like that at all, because that's tedious boilerplate few will bother writing. Instead, the constraints will be left completely unstated and depend on internal team knowledge / humans not making mistakes in order to not explode (which means... it WILL explode)

Something about how we combine those attributes together causes the meaning as a whole to get lost. We find ourselves in a similar spot to where we started in chapter 2,

the meaning of our data types only really exists ephemerally inside of the constructor when the validation code is executing. Once it's done, we're left with a data type that "forgets" parts of itself.

This representation, with all of its flaws (including the multiple Booleans), is one that existed in a real application for real rockets (though, anonymized and simplified to protect the guilty (me)). This problem isn't unique. It has existed in every code base, in every language, and across every team on which I've ever worked. It's pervasive because it happens in real applications the same way it did here: through the slow accretion of "easy to add" features. Requirements show up that boil down to "we want the existing thing to do just one more thing", and the code required to do it is "small" – just another field or two, so we add them without taking a step back and asking what effect these additions have on the meaning as a whole.

However, the good news is that these problems can be avoided once you learn to recognize them! We're already half-way there. We know something is off. We're doing all this defensive programming because the meaning of our code is out of line with the meaning of our data. What we have to learn to see is where the two began diverging, and how we can bring them back.

A QUICK RESET

Let's do something that happens all the time in real life: reset and try again. Our DRY focused refactoring didn't give us the effects we wanted (which is fine!). So, let's rewind the clock to undo that change.

Further, let's rewind even further to go back to when we first encountered the completedOn and completedBy attributes. This was when we first started getting that feeling of friction in our design. Things only got worse from there when we introduced the notion of skipping. So, we'll start over from that point, and see if we can point the ship in a better direction. Once we get our trajectory right, adding things like Skipped back in should be a breeze.

So, for the next section, we'll be starting from here:

Listing 4.23 Where we're starting.

```
record Status(  
    Template template,      #A  
    Step step,              #A  
    boolean isCompleted,    #B  
    User completedBy,      #B  
    Instant completedOn,   #B  
){/*...*/}
```

#A just our initial attributes

#B along with completed on/by. We'll add skipped back in once we've sorted out our modeling

4.2 Obvious things which maybe aren't so obvious (Part II)

There's something that's sort of obvious in a "yeah, duh" way, but that's also, maybe not so obvious until our attention is called to it: there is an implicit `AND` sitting between each attribute we define on a record.

Listing 4.24 What's currently hidden in our model

```
record Status(  
    Template template,  
    AND          #A  
    Step step,  
    AND          #A  
    boolean isCompleted,  
    AND          #A  
    User completedBy,  
    AND          #A  
    Instant completedOn,  
){/*...*/}
```

#A When we define a data type, we're saying it's made up of attribute 1 AND attribute 2 AND ... AND ... AND.

Which is, like I said, pretty obvious. However, the effect these `AND`s have on us might be less obvious. Their influence is subtle, yet, if we're not careful, corrupting. Without caution, they can cause us to accidentally shift where we express the meaning in our programs. This is what happened to us when we introduced the `completedOn/by` family of attributes. We stopped expressing our meaning in *code*. Instead, we started baking the meaning into the "English prose" that lives *in* the code.

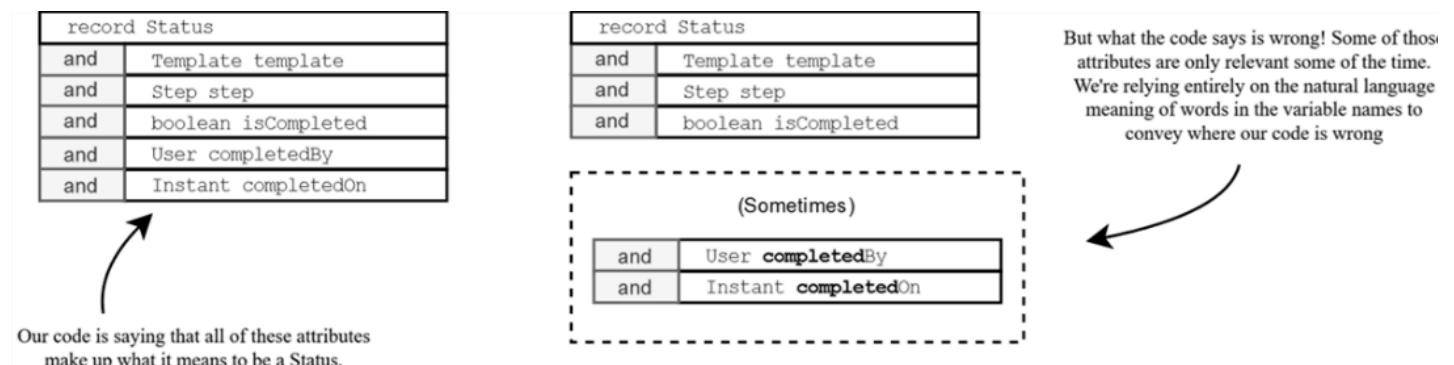


Figure 4.5 What our code says versus what it actually means

As a result, our code stopped describing what it *is*. Actually, it's a bit worse than that: it started describing something *other* than what it *is*. As we designed our status type, we were trying to model something that involves multiple ideas: a checklist item that's not started, *or* checklist item that's completed, *or* a checklist item that's skipped. However, our code doesn't say anything at all about those different options. There's no *this or that*. There's only `AND`, `AND`, `AND`. Which is why there's so much friction between what our code *says* a `Status` *is* versus what we know a `Status` *is supposed to mean*. The design of our code, with its `completedOn` AND `completedBy` attributes, really only expresses

exactly one of those many possible states: a checklist item that has been completed. If you read the code exactly as it's written, that's exactly what it says (why else would it have those attributes!). The other states, like "not started," are achieved by using nulls to bludgeon past what the code says and hoping for the best.

Listing 4.25 Reading the code for exactly what it says

```
record Status(          #A
    Template template,
    Step step,
    boolean isCompleted,
    User completedBy,      #B
    Instant completedOn,   #B
){/*...*/}
```

#A This collection of attributes really represents a Completed Checklist item, not some general Status
#B What else would this record be modeling if not completed step? What else would these "completed" attributes be doing here!

This is the trick that the ANDs play on us when we're not paying attention. They convinced us to stick a bunch of attributes together that don't actually go together. Why would we have completedOn there if the Status isn't actually completed yet? The code starts expressing the wrong thing, so we're then forced to start side-channeling what we actually mean elsewhere.

Suddenly, the *prose* we put into our names starts pulling double duty. Words like "completed" in "completedOn" become more important than the code itself. They have to do all the work of conveying to the reader that what's written in the code is wrong. We're entirely relying on our readers to "get it" when they find key words like "completed." We expect them to understand a completely unstated expectation that the code should be interpreted in a special way – and one which directly contradicts what the code says.

We force readers of our code to parse it along multiple semantic dimensions. There's the initial reading of the code as code, but this is, of course, incomplete or misleading. So, they have to read the code again, but this time as English prose to see if they can pull out any contextual clues which can patch where the code is incorrect.

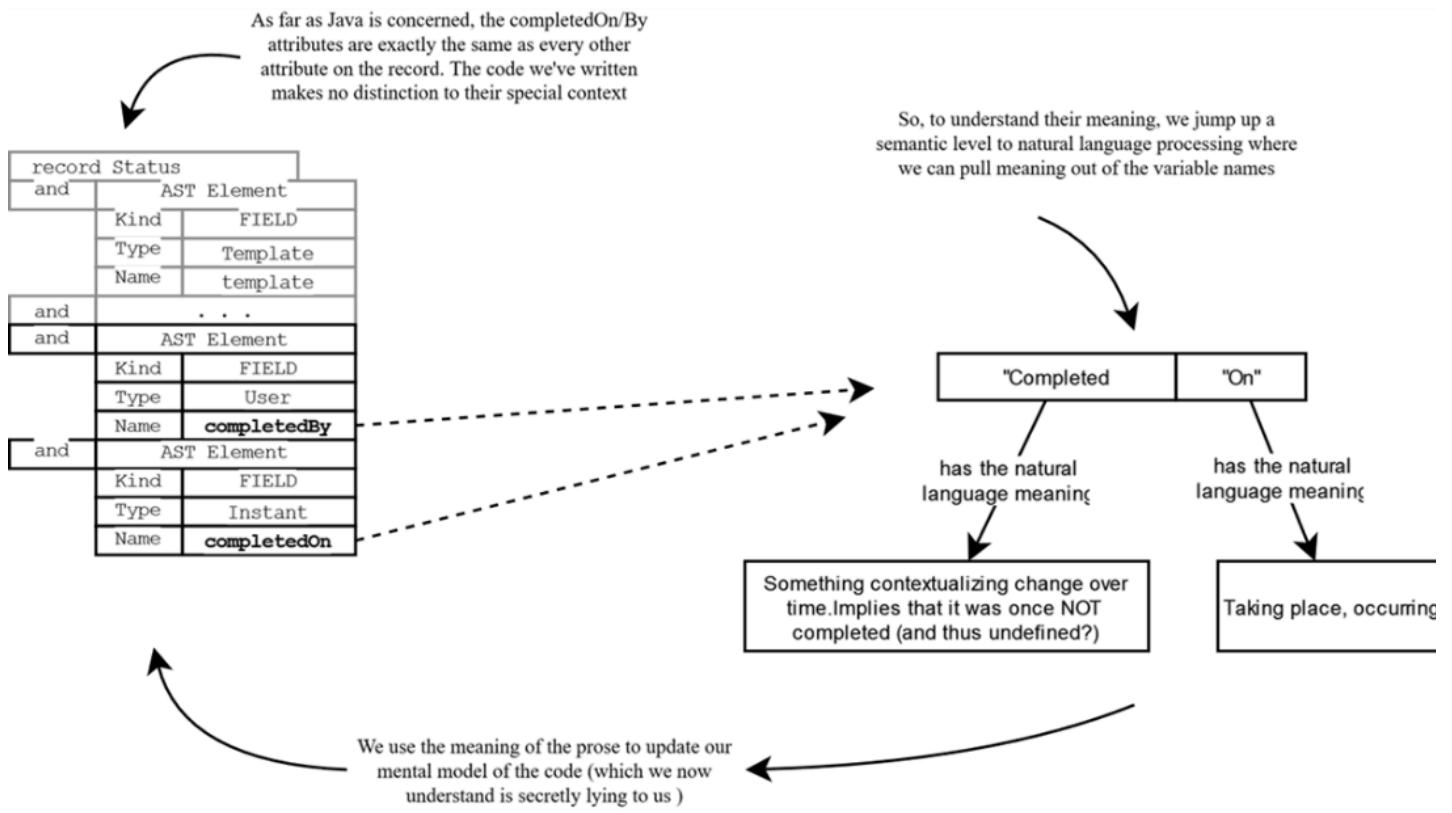


Figure 4.6 Approximating what we actually mean from multiple semantic readings of the same code

This is not the kind of code we want!

Whenever I get stuck down paths like this during the design process, it's usually when I take a step back and try to pare down to what the core thing we're talking about is. Usually this is like, "Ugh. Ok. What is it that I'm *actually trying to say?*". There's something that exists in the data that doesn't exist in the code. What is it that we're missing?

It's here that we can fall back to the simple exercise we introduced in chapter 2: comparing the meaning of our data with the meaning of our code. This is our guiding light through the darkness. The "design process" is largely just bringing the two worlds into alignment.

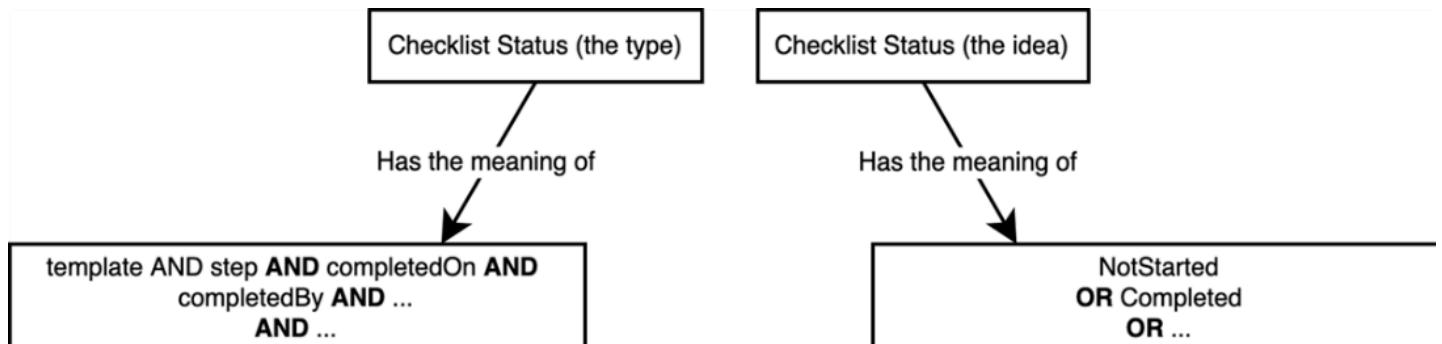


Figure 4.7 comparing the conflicting meanings between our data and our code

A checklist status isn't one single thing with a sea of nullable, sometimes defined attributes; it's a *multitude* of top-level ideas, each described by their *own* set of relevant attributes. This is what's missing from our modeling.

What's really interesting here is that AND changes once you start paying attention to it. If it's implicit and in the background, it puts a negative downward pressure on our design process. It causes us to combine attributes which don't belong together. However, if we design consciously, while paying careful attention to what it means to AND two attributes together, it becomes a *positive* upward pressure on our designs. It can help us discover when we've gone off track earlier in the design process. We just have to learn to listen to the feedback it gives us.

4.2.1 Designing consciously with AND

The only trick here is learning to read the code for exactly what it says. The big allure is to do what we did in the previous sections: rely on English prose readings of the code to supply the meaning. Instead, we'll read the code as though we were the Java compiler. For exactly what it says as written. And no more. This is how we use AND to give us feedback on our design.

So, let's take another stab at designing our Status data type. Just as before, we know that it's "natural key" is formed by the template and step for which it is tracking the status. We know combining those attributes is a step in the right direction.

Listing 4.26 going piece by piece

```
record Status(  
    Template template,      #A  
    AND Step step          #A  
){}  
#A So far so good. These ANDs make sense
```

Next, we introduce another attribute. Again, paying close attention to what the combination of these attributes means for our data type as a whole when combined together.

Listing 4.27 Testing the waters with isCompleted

```
record Status(  
    Template template,  
    AND Step step,  
    AND boolean isCompleted,  #A  
){}  
#A This is kind of OK, but... we know what's coming next
```

Now we get to our first contextual attribute: completedBy. This is where we hit the first major issue.

Listing 4.28 Combining this attribute breaks the meaning of our data

```
record Status(  
    Template template,  
    AND Step step,  
    AND boolean isCompleted,  
    AND User completedBy      #A  
){}  
#A And right here we hit a hard wall. This attribute cannot be ANDed with the rest, because it's only  
defined *sometimes*
```

This attribute immediately causes friction. But why? What is it conflicting with? The other attributes? Not really. If you combine a Template, a Step, and CompletedBy, you have something that sounds perfectly valid. In fact, it sounds a lot like... the exact data we would have for describing a Status that has been *completed*.

The friction is with *where* we're putting these attributes and what we're wanting it to *mean*. This combination of attributes is only a problem if they're being combined under a data type called "Status," because they don't describe a generic Status – they describe a *completed* status! So, let's allow the ANDing of these attributes guide us. Rather than fighting it, let's embrace the feedback it's providing us: Completed is its own data type.

Listing 4.29 We've been describing a Completed data type

```
record Status Completed(          #A
    Template template,
    AND Step step,
    Boolean isCompleted,          #B
    #A
    AND User completedBy,
    AND Instant completedOn,
){                                #C
}
```

#A We're not modeling a general Status, our attributes describe a Completed status, so let's honor their meaning and change the name

#B We can get rid of this boolean, because we're now modeling exactly one thing

#C Note that the record's body is totally empty. We don't have anything illegal states or ambiguous nulls to defend against any more!

Ok. Do you feel that? It's like things suddenly snapped into focus. Everything that was wrong with our previous model, where we were storing completedOn and completedBy on a record called Status, immediately disappeared when we stopped fighting against what this combination of attributes wanted to express – that it represents something that is Completed.

This, to me, is one of the best moments during the design phase. It's what makes spending all of this time massaging the data and pursuing its meaning worth it. It feels a bit like unlocking something that was hidden. Checkout the body of our record in listing 4.29. That used to be filled with defensive validation code that existed only to guard against the nonsensical states that our modeling allowed. Now it's totally empty! Those illegal states have been completely eliminated.

But, of course, this is just one of our checklist states. And having a checklist item that's only ever Completed doesn't make sense. We need to model the remaining states. What's cool is that we've unearthed something fundamental about *how* to model the remaining items in our domain. We know from all of our work so far that there's no way to stick that information onto the *same* record by just ANDing things together. If we try, we'd end up right back where we started: nulls, illegal states, and defensive programming. So, our modeling hand is forced, we have to create a new type.

Listing 4.30 When a checklist isn't yet completed, it is...

```
record NotStarted(
    Template template,      #A
    AND Step step          #A
) {
}                                #B
```

#A ANDing the only attributes which make sense for the initial state of our checklist. There is no mention of completedBy here, because it's not completed! It's Not Started!

#B Same deal as the listing above. Totally empty! Nothing to defend against, because we've modeling the meaning of the data

NotStarted can only have those two attributes, because that's what it means to be Not Started! Because we're not including attributes that don't belong, we're able to smoothly add these other states. And again note: no defensive programming needed in the constructor.

ADDING SKIPPED BACK INTO THE MODEL

With our original modeling, each time we added a new status to our checklist, it was a stressful ordeal. We kept ending up with *new* illegal states that needed to be defended against, and those defenses needed tested, and the meaning of those new states had to be conveyed "outside" of the code through naming hints.

With our new modeling we can freely add more and more status types for our checklist without any conflict. This is how we use AND to our advantage. We let it push us towards modelings that group the right attributes together. This new awareness makes adding the Skipped status a breeze.

Listing 4.31 Reintroducing the Skipped status using the new modeling

```
record Skipped(           #A
    Template template,
    AND Step step
    AND User skippedBy   #A
    AND Instant skippedOn #A
    AND String rationale  #A
) {                      #B
}
```

#A The combined meaning of the attributes ANDed together is in perfect alignment with the meaning of the record as a whole

#B Same as before: nothin' here.

This is the power of data modeling. We've made illegal states impossible to even express in the code. They've been "deleted" from the code as a whole. They just don't even exist anymore.

A NOTE ON THE IMPLEMENTATION

This is one of the many, many, many ways we could approach representing these data types in Java. Is this The Official Way to do it? “It depends.” Translating a model into Java involves design work (and a fair amount of intuition and experience). We have to decide which aspects to highlight versus which to hide away. The approach we took here is simply *one way* of doing it. We went down this path because it lets us explore some interesting modeling aspects of Java. The important part is not the specific implementation we chose, but that the implementation successfully encoded what our data means into Java. As long as we satisfy that, the exact “how” comes down to what makes sense for your particular application, its use cases, and a fair amount of personal taste. Throughout the book, we’ll look at alternative ways of modeling these same ideas which bring different benefits, tradeoffs, and ergonomics.

4.3 This OR that Or...

Our individual statuses are well modeled, but they’re currently just sort of floating out there in the general ether which makes up our domain. We “know” that NotStarted, Completed, and Skipped are all Checklist Statuses, but nothing in the code ties them together.

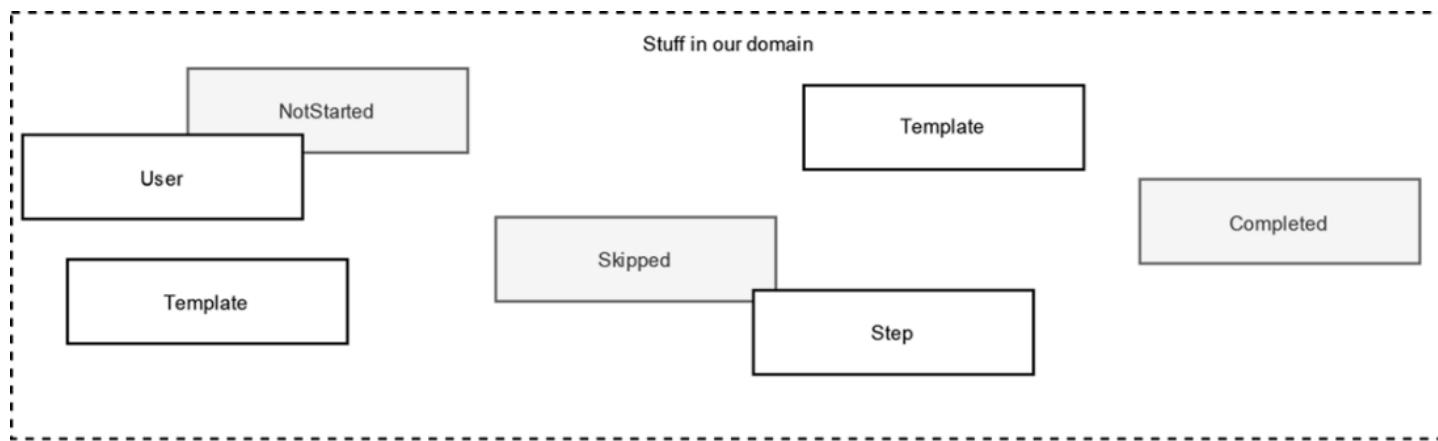


Figure 4.8 the Statuses are well modeled, but nothing ties them together under a single idea

What we want is to bring them all together under a single canonical data that denotes that these records are all related, but represent different options within a single idea (this OR that OR...).

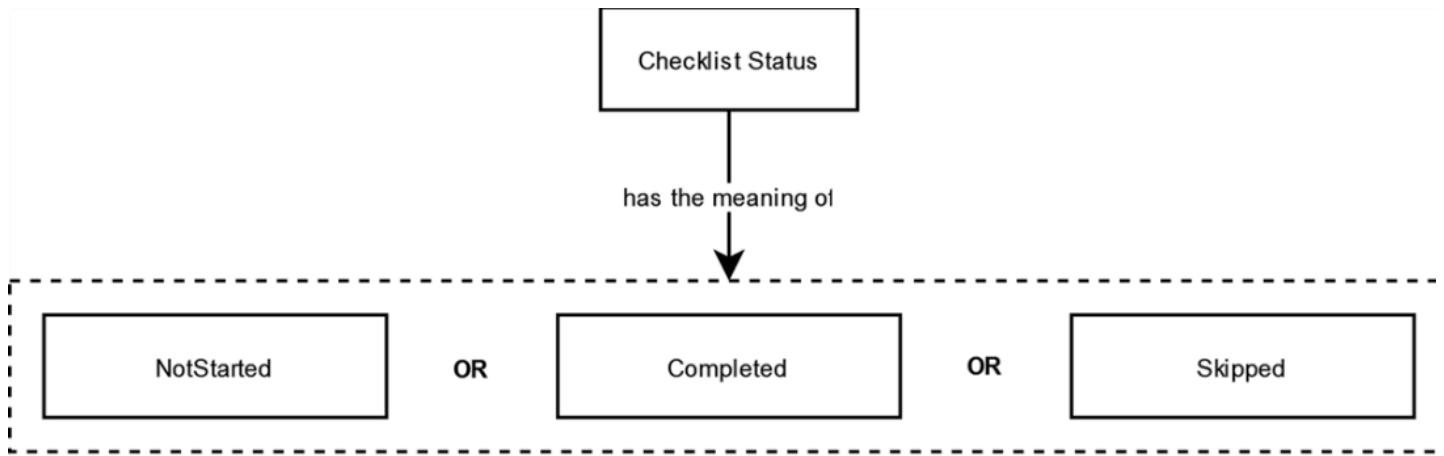


Figure 4.9 What we want

4.3.1 Interfaces as data types

Interfaces are usually associated with behaviors. They represent a contract that the implementing code promises to satisfy. However, interfaces can also be used for things other than behaviors. They can be used as *data*.

Listing 4.32 Defining a new data type with an Interface

```
interface ChecklistStatus { #A
    #B
}
```

#A The interface name is treated a data type in our code

#B The body of the interface is empty, because we're not defining any behaviors.

What can we do with an interface that defines no methods? We can implement it as usual!

Listing 4.33 Implementing the ChecklistStatus interface

```
record NotStarted(
    Template template,
    AND Step step
) implements ChecklistStatus {} #A
```

```
record Completed(
    Template template,
    AND Step step
    AND User completedBy
    AND Instant completedOn
) implements ChecklistStatus {} #A
```

```
record Skipped(
    Template template,
    AND Step step
    AND User skippedBy
    AND Instant skippedOn
    AND String rationale
) implements ChecklistStatus {} #A
```

#A Making each of our related record types implement the interface

When we do this, we're treating the interface not as something that gives us behavior, but as something that gives us new data. Its *meaning* is no different from a record: it represents a piece of data in our domain. The records that implement it represent the different options this new data type can be.

It allows us to express in our code that a Checklist Status is either NotStarted OR Completed OR Skipped.

Listing 4.34 Capturing the statuses as types

```
ChecklistStatus notStarted = new NotStarted(
    template,
    step
);

ChecklistStatus completed = new Completed(
    template,
    step,
    new User("Bob"),
    Instant.now()
);

ChecklistStatus skipped = Skipped(
    template,
    step,
    new User("Bob"),
    Instant.now()
);
```

This setup is a lot like creating a fancy enum.

Listing 4.35 Comparing enums with interfaces and records

```
enum ChecklistStatus = {
    NotStarted,
    Completed,
    Skipped;
}

interface ChecklistStatus{}
record NotStarted(...) implements ChecklistStatus {}
record Completed(...) implements ChecklistStatus {}
record Skipped(...) implements ChecklistStatus {}
```

We're creating an *enumeration* of the possible values our ChecklistStatus can be. The big departure from enums, and what gives us so much power, is that records allow us to attach arbitrary attributes to each of our enumerates states. We're not restricted to fixed constants like we are with Enums.

4.4 Open or closed?

There's one minor gap remaining between what the code in our Java implementation says versus what our data is supposed to mean. A Checklist status in our domain is *closed*. It is defined to be three things (NotStarted, Completed, Skipped), only those three things, and *nothing else*.

Listing 4.36 Our closed definition of Procedure

```
Procedure ==  
  NotStarted  
  OR Completed  
  OR Skipped  
  #A
```

#A Note that it doesn't say "or, y'know, anything else you want it to be"

This is an extremely important aspect of modeling because it allows us to reason exhaustively about the surface area which makes up our domain. Sometimes we want extensibility and the ability to grow the behavior of an API over time through extension and specialization. Libraries are a good example of this. We're almost always fighting to ensure they're flexible and open enough to support use cases we haven't considered. However, just as often (if not more), and especially in application design, we don't want things to be open. We want them to reflect the thing we're modeling.

Openness gets championed as a universally good property (it's the "O" in the hallowed "SOLID" principles, after all). However, as with all things engineering, we need to remember to do the *engineering* part. We have to weigh if the properties under consideration make sense for *our* system and *our* domain. Blind application of "universal good" is cargo culting.

Listing 4.37 Are these relevant to our domain?

```
record Blocked() implements ChecklistState {}  
record Paused() implements ChecklistState {}  
record Started() implemetns ChecklistState {}
```

These extra states in Listing 4.37 might be perfectly valid in some other Checklist related domain, but *they are not valid in ours*. We don't want these states in our system. Or if we do, we want to be very explicit about how and when they're introduced.

The problem with our current Java implementation is that anybody can come along and implement our interface and inject a new idea into our domain. We've haven't truly captured what our data means, because while the data's meaning is closed, our java design is open.

Luckily, Java 17 introduced some new tooling in the language to close this gap.

4.5 Representing Sealed Families of Data

Sealing enables us to be explicit about what is permitted to subclass or implement our types. It closes down future extension of unknown types. This again, might sound very uncomfortable at first. Maybe even wrong. Openness is beat into us as developers. But the ability to say what something *is not* is an important tool for representing what something *is*. If you can buy that Enums are a valid modeling tool despite being "closed", you'll also buy that data types themselves have lots of reasons for being closed to future extension. Sealed classes are essentially Enums that can store extra stuff.

We can create closed families of data in Java by using the new `sealed` and `permits` keywords. Let's update our Procedure definition

Listing 4.38 Sealing the ChecklistState type

```
sealed interface ChecklistState          #A
  permits NotStarted, Completed, Skipped #B
  {}

record NotStarted(/*...*/) implements ChecklistState {}; #C
record Completed(/*...*/) implements ChecklistState {}; #C
record Skipped(/*...*/) implements ChecklistState {}; #C
```

#A Adding the “sealed” modifier to our interface definition

#B And then explicitly specifying what’s allowed to implement it

#C These are assumed to be in their own files

Sealing prevents any future classes or records from implementing our interface.

Listing 4.39 Trying to extend the interface

```
record Blocked() implements ChecklistState {} // ERROR #A
```

#A Nope! Java will give a compiler error that Blocked is not part of the sealed hierarchy

How cool is that!

SKIPPING THE PERMITS CLAUSE

The `permits` clause is only required if you have your records broken out into their own files. If you declare all of your records inside of the interface's body, you can skip the `permits` part of the definition all together. Java will implicitly treat whatever's defined in scope as being the set of permitted values.

Listing 4.40 Sealing without an explicit Permits

```
sealed interface ChecklistState { #A
  record Pending() implements ChecklistState {};           #B
  record Completed(/*...*/) implements ChecklistState {}; #B
  record Skipped(/*...*/) implements ChecklistState {};    #B
}
```

#A We don't list out the explicit permits clause

#B Defined in the interface's body.

It's a small change, but a pretty delightful one to my eye. It drives home that these are all one logical *thing*. The look and feel of the code directly mirrors the meaning of the data it's modeling. The code says, in one spot, that a checklist status in our domain is these three things, only these three things, *and nothing else*.

4.6 Algebraic Data Types

The records and sealed interfaces we've explored in this chapter are Java's version of a special category of types called Algebraic Data Types. They get the “algebra” in their name from the fact that these types can be manipulated symbolically just like we do numbers. Algebraic types have operators (just like numbers!) and a set of laws that

hold for how those operations must behave (just like numbers!). Everything we're doing up in Java land, with all of its syntax around records and sealed interfaces, is just the "implementation" of this algebra. And the whole thing consists of just two operators: + and ×.

So, we're going to take a quick look at this algebra and how its operations relate to our Java records and interfaces. Dipping down into this world lets us see what's actually going on in Java behind the scenes when we make new data types. It gives us a concrete semantics for what AND and OR mean when applied to types. Knowing the basics unlocks some additional ways for us to analyze and talk about our programs. Plus, it's kind of cool.

4.6.1 Product Types

Records are Java's implementation of what are called Algebraic Product Types. These Product Types, just like records, create a new type by combining other types together using the semantics of AND. However, whereas records define this combining with all kinds of syntax, keywords, names, and curly braces, we define a new product type in our algebra by simply applying the "product" operator (×) to the types we want to combine.

Table 4.1 comparing creating a new type with records with creating a new type with products

Java Record	Algebraic Definition
enum Drinks {Coffee, Tea}; enum Served {Hot, Cold}; record Beverage(Drinks drink, Served served){} 	Beverage = Drinks × Served

To understand what the algebra is doing, and why these are called "Product Types," we have to go back to our toolkit from Chapter 3 where we see "through" our types into the concrete sets of values that they denote. All types are inhabited by a set of values. For instance, a boolean ultimately represents the set of values {True, False}, similarly a Byte is "made up of" the integers from 0-256. We've used this set based view of the world as one of our main tools for analyzing if the representation of our types is in line with our domain's meaning. However, this view has more benefits. We can use it to explore what's really going on at a deep level when we say "combine this type AND that type."

So, if we mentally swap out "type" for "a set of values," then the "product" part of the name suddenly becomes a bit clearer. We produce new types in our Algebra by taking the *Cartesian product* of the set of values that backs each of the types being combined. In the algebraic view, we're producing a new type which is inhabited by the set of all pairs of the types we've applied.

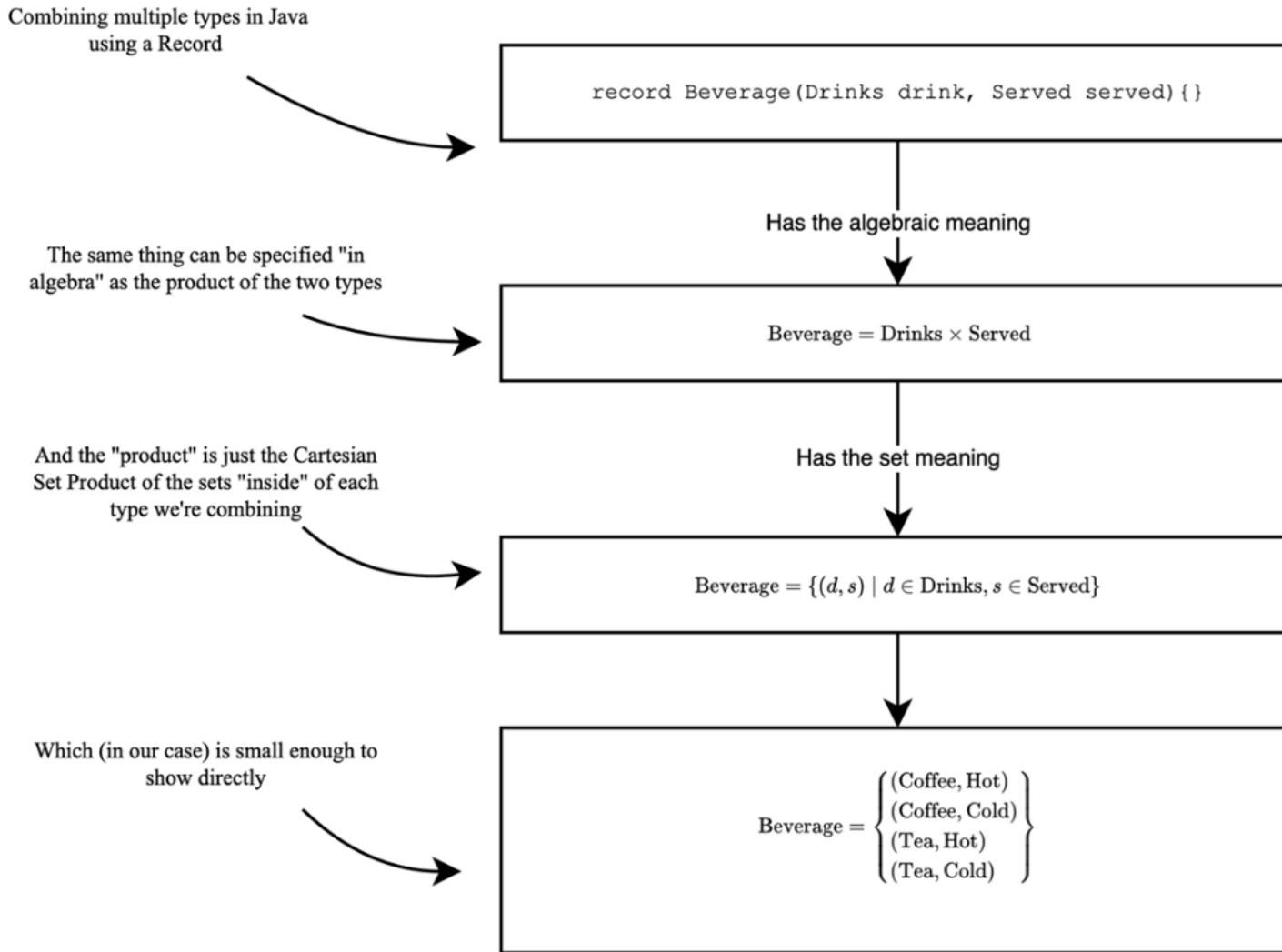


Figure 4.10 Viewing Records and Product Types as the cartesian product of sets

This isn't just an academic definition; we can see the cartesian product in action on our records. It directly determines the set of possible states that we can create with our new type.

Listing 4.41 Every possible state we can create with our new type

```

new Beverage(COFFEE, HOT);      #A
new Beverage(TEA,   HOT);      #A
new Beverage(COFFEE, COLD);     #A
new Beverage(TEA,   COLD);      #A
  
```

#A The “state space” of our new type is the set product of the types from which it’s built.

Thinking about what it means to combine things together gives us another vantage point from which to judge our data modeling. The fact that behind every record we define is a full cartesian product of each of its attributes is kind of sobering. It's why ANDing things together without caution (as we did earlier in this chapter) can cause our designs to so rapidly go off the rails. The total number of possible states we can express grows by a multiplicative factor with each type we add! It doesn't take long for types to become massive unwieldy things with way more invalid states than valid ones.

4.6.2 Sum Types

Sealed Interfaces are Java's implementation of what are (usually) called Algebraic "Sum Types." (You'll also see them called by other names like "Tagged Union", "Variant", or "Coproduct" types). These types are all about what it means to combine types with the semantics of OR. They let us represent choices *between* multiple types in our code. And just like we saw with Product Types, Sum Types create new types with a very simple operator: +

Java Sealed Interface	Algebraic Definition
sealed interface Drinks { record Coffee() implements Drinks{}; record Tea() implements Drinks{}; }	Drinks = Coffee + Tea

Now, unlike Product types, which were easiest to think about in terms of sets, the most intuitive way to connect where the "sum" part of the name comes from is to stay up in the world of types, and think about how many types are "contained" within the sealed interface you're defining. The number of types we can choose from is the literal count – the *sum* – of how many types we've sealed.

Listing 4.42 Counting how many types are sealed under our interface

```
sealed interface CountVonCount {  
    record One() implements CountVonCount {} #A  
    record Two() implements CountVonCount {} #B  
    record Three() implements CountVonCount {} #C  
}  
#D three options!
```

#A The new Sum Type we're creating
#B and it has one...
#C two...
#D three options!

And that's pretty much it – or that's enough for our purposes anyways. Product Types give us AND semantics while combining types, and map to Record Classes in Java. Sum Types give us OR semantics while combining types and map to sealed interfaces in Java land.

Table 4.2 Mapping between the Algebraic and Java representations

Java	Algebraic Type	Algebraic Operator	Combines via
Record	Product Type	x	AND
Sealed Interface	Sum Type	+	OR

Despite their simplicity, product and sum types give us enough power to model just about anything we can imagine. All of it built just from two simple rules for what it means when we combine types together. Our examples have all been small throughout this chapter, but we can create structures of arbitrary (sometimes recursive) complexity

without ever needing anything other than these basic types. They give us the tools to build big things just by stitching together smaller ones.

WHAT ABOUT THE SET VIEW OF SUM TYPES?

The “count how many types are sealed” approach is a good enough intuition for day-to-day usage of Sum Types. We’re sticking with that one because viewing what’s going in the sets behind the scenes gets messy enough to put it beyond the scope of this book. Sum Types are generally represented as sets via a disjoin union (which is why you’ll often see these types called “Union” or “Tagged Union” types). While there’s nothing terribly complicated, it does take some extra “machinery” (in the form of functions) on top of the sets to map between different representations. If you’re up for some extracurricular activity, mapping between Java, the algebra, and the set representations can be a fun programming exercise to explore.

4.6.3 What’s the point of knowing this stuff?

Why bother looking at any of this algebra stuff? What’s the point when it only really “exists” as a set of abstract ideas?

The first is just familiarity. Even though we’re Java programmers, not everyone else is. A “sealed interface” only means something within the context of our specific language. This is true of all implementations across all programming languages. These algebraic ideas will be wrapped up in the specific syntax, historic design decisions, and idiosyncrasies of their host language. What unites us all – what gives us a “ubiquitous language” (to steal some Domain Driven lingo), and allows us to speak to each other, is the underlying algebra. You’ll see Product and Sum Types widely referenced in the outside world. Knowing what they are and how they relate back to Java lets us all talk about these patterns and learn from each other without the impedance mismatch that would otherwise occur if we had to speak in terms of each programming language’s specific implementation.

The other big benefit, and this might just be my favorite, is that understanding the algebra gives us an alternative and super terse language to “speak in” while sketching out a problem at a whiteboard. Its terseness allows us to pack a ton of information into a dense area, which we can’t do if we try to communicate or think “in Java.” I’ve always found that the more information I can see, the easier it is to play around with the ideas and notice connections between things while modeling.

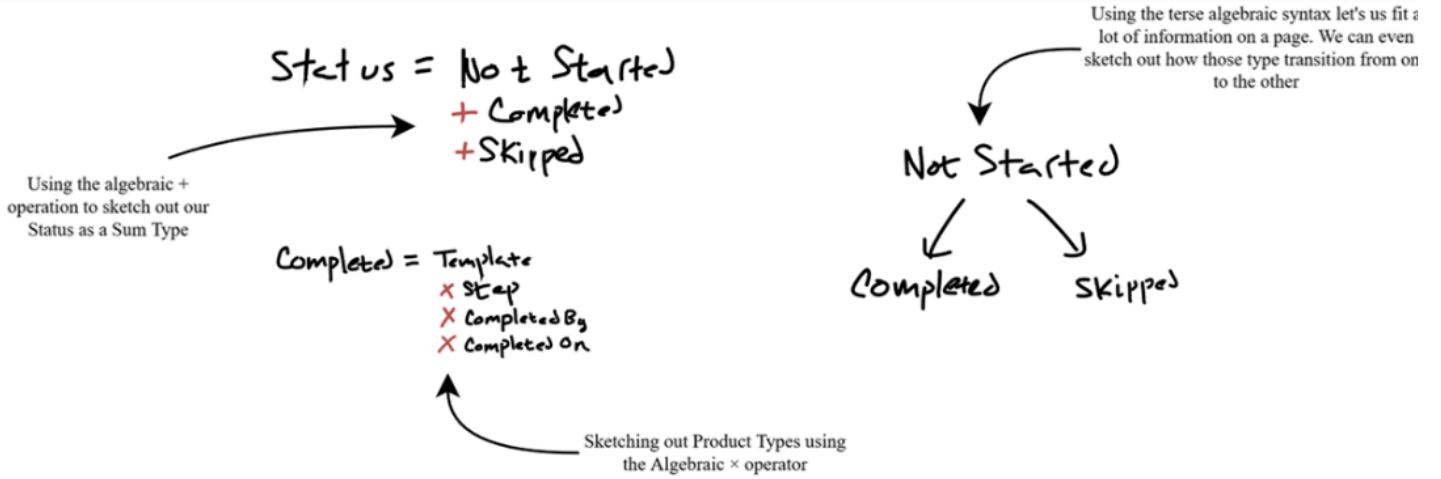


Figure 4.11 algebra as a sketching language

Further, once you know what the algebra means, you can invent your own syntax for its operations. For instance, one big problem I have with the normal syntax of Algebraic Data Types is that, no matter how much I try, I will always read the $+$ sign as “combine these things with AND,” rather than what it actually means, “combing these things with OR.” But that hang up isn’t a big deal, because it’s the *semantics* of the algebra that matters. The symbols we use to express it are just arbitrary bits of syntax. As long as we know what they mean, we can replace them with any other arbitrary bit of syntax that we like! For my brain, which thinks in terms of “AND” and “OR”, rather than “Product” and “Sum,” I drift towards using the associated logical symbols ((AND) \wedge and (OR) \vee) which directly map to those ideas.

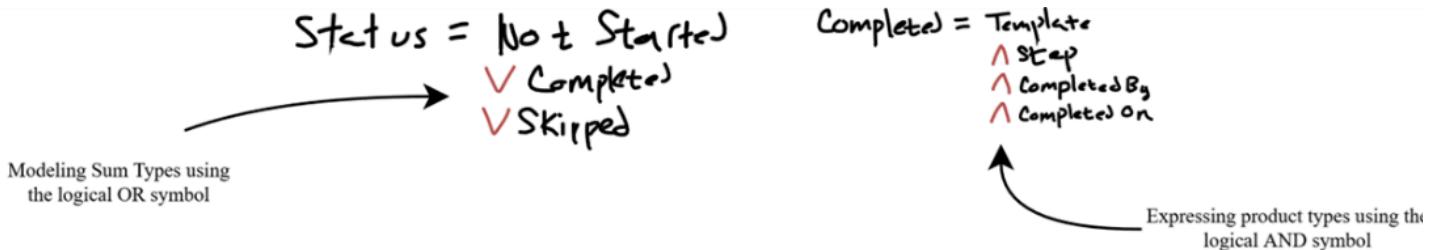


Figure 4.12 using different symbols doesn't change the underlying semantics

Wrapped up in this simple algebra is the core idea that permeates the book: meaning and semantics transcend everything else. If we really understand the semantics of what we’re modeling, we buy ourselves a lot of freedom for thinking “above” the specific implementation details, language, and syntax. We can even change those implementations entirely with no effect on our program as a whole, because we’re operating on something more bedrock and fundamental. The guiding light is always: “are we capturing the meaning?” If so, the specifics mostly take care of themselves.

4.7 What to do if you're on older versions of Java

The ability to express the data in our domain using algebraic types is one of the coolest new features in Java. However, at the end of the day, it's important to remember that they're *additive*. Sum types (via sealed interfaces) are just a way of telling the compiler that one part of our domain is fixed to a particular set of known things. The important

part is the modeling -- the fact that we've taken the time to figure out the distinct things which make up our domain and how to represent them with data. Enforcement by the type system, when available, is just icing on the cake.

So, while we don't have the compiler's help in older versions of Java, we can still cue readers of our code that our *intention* for this group of classes or records is to act as though they're sealed. You can do this with big attention grabbing "Hey! Readme!" Java doc headings, or hand rolled informational annotations like (`@Sealed`), but I think the best way is to just move all of the related types into the same file (just as you would do with actual sealed types if you wanted to avoid explicitly specifying the `permits`)

```
interface ChecklistState {  
    record NotStarted(/* ... */) implements ChecklistState {}  
    record Completed(/* ... */) implements ChecklistState {}  
    record Skipped(/* ... */) implements ChecklistState {}  
}
```

It doesn't *do* anything -- Java has no idea what it means without the sealed qualifier. But, for the people using and writing the code, this convention can serve as a good marker to drive home how you're intending the class to be used.

4.8 Wrapping Up

In this chapter we took a long tour through how we might design checklists from a data-oriented perspective. Though our missteps, we learned to identify when the code had been lured into expressing something other than what it intended. These often show up as illegal states that exist solely due to the modeling we've chosen. It surfaces as a "friction" while writing the code. We resolve it by using our tools from chapter 3, comparing meanings. Getting the code to capture the meaning of the data its modeling is our light through the darkness.

Further, we learned to lift implicit ANDs up to the surface. When we design with them consciously in mind, they become a powerful tool for pushing our data modeling in the right direction. If we ignore them, their implicit nature can subtly nudge us towards expressing things we don't mean.

Lastly, we learned how to use records and interfaces to express choosing between one of many different options with OR. This pattern is like enums, but on steroids. It lets us express enumerated sets of values, that each contain complex runtime values (as opposed to the fixed constants enums to which enums are restricted)

4.9 Summary

- "Representation is the essence of programming"
- In data-oriented design we focus on the representation of the data and what it means rather than the state and its operations
- Design is an iterative and messy process (and that's OK!).
- Defensive programming is a warning sign that the meaning of the code might be out of line with the meaning of the data

- We have to train ourselves to become sensitive to when we start relying on English prose that lives in the code to convey to readers where the code itself is wrong
- There is an implicit AND between every attribute we define on a record
- If we don't pay attention, we can AND attributes together that have conflicting meanings and cause our code to lose focus
- When we design consciously, we can use AND to push our data modeling in good directions.
- We have to be on the lookout for when we're using AND instead of OR
- Interfaces can represent more than just behaviors. They can represent data types
- Sealed classes and interfaces allow us to express "these options and no others"
- Records and Sealed Interfaces are how we represent Algebraic Data Types in Java
- Product Types are built with the \times operator and combine type with AND
- Sum Types are built with the $+$ operator and combine types with OR

5 Modelling Domain Behaviors

This chapter covers

- Modeling complex behaviors as pipelines
- Letting data guide the design
- Making illegal Behaviors impossible to represent

We're going to leave the cozy and safe worlds of oil curing and checklists and jump head first into the messy, frantic world of software development. No more isolated modeling. We're going to take the data-oriented tools we've learned and apply them to building a complete feature.

This means dealing with everything that makes software development hard. Databases, frameworks, services, and, worst of all, *existing legacy decisions*. Everything in this chapter will be done inside of an existing application (one we'll make up as we go). We're going to learn how to carve out a world built on data inside of an existing one built on mutation and identity objects.

This chapter will be challenging. The feature we're going to implement is non-trivial and built from complex requirements. At the end of this, you'll be battle hardened and have all the tools needed to tackle any feature in your own applications.

Let's get started!

5.1 "Encouraging timely payments"

We're going to explore the spiritually fulfilling world of invoice processing and charging late fees. (Or, as it's known in corporate speak, "implementing systems which encourage timely payments.") Customers aren't paying their bills on time, and the accountants can't keep up, so we're going to automate their process.

In the previous chapters, we walked through the process of gathering requirements in detail. It remains the most important step, but we'll pretend we already did it so we can focus on the implementation side. Here's what we've got as a set of formal(ish) requirements:

Table 5.1 Requirements!

ID	Requirements	
A0	The system shall charge late fees to customers with open past due invoices as of the Evaluation Date plus the customer's Grace Period	
A1		The fee shall be calculated as a percentage of the customer's total past due
A2		Fees shall be computed in USD
A3		The Due Date of the Fee Invoice shall be based on the customer's Payment Terms
A4		Delivery and Invoice Number generation shall handled by the Billing System
B0	The customer shall have a Grace Period	
B1		Customers in good standing receive a 60-day grace period
B2		Customers in acceptable standing receive a 30-day grace period
B3		Customers in poor standing must pay by end of month
C0	The system shall charge these fees based on Configurable Rules	
C1		The system shall have a Minimum Due Threshold
C2		The system shall have a Maximum Due Threshold
C3		The Fee Percentage must be configurable based on the customer's billing Country Code
D0	The system shall not charge late fees for special cases	
D1		Fees less than Minimum Due Threshold shall not be sent to the customer
D2		Fees greater than Maximum Due Threshold shall not be sent to the customer UNLESS the customer has been marked as Approved for Large Fees
E0	The system shall track Customer Approvals	
E1		Customers exceeding the Maximum Due Threshold must be flagged for Manual Approval
F0	All fees will be auditable and recorded by the system	
F1		The system shall record which invoices were involved in the fee computation
G2		The system shall record a rationale for invoices which cannot be billed

Summed up in plain English: we find all the invoices for customers that are past due, add up their outstanding balances, then cut a new invoice with a fee that's a percentage of that total past due. Easy!

The hairy parts are in the details. Due dates have all kinds of special rules and exceptions. And half of the requirements deal with figuring out if this fee can even be sent to customers. There's a narrow goldilocks zone. Too small and we skip it. Too large and it needs manual approval. On top of all that, every decision needs to be auditable. So, our fees need persisted whether we bill them or not.

To further complicate things, we're going to embed this feature into an existing application that's already steeped in design choices we didn't get to pick (or maybe picked ourselves, but now slightly groan at in hindsight). There's already a database, a very opinionated framework, an ORM (or other "magical" annotation-based mapper), and a host of service integrations with APIs we don't control.

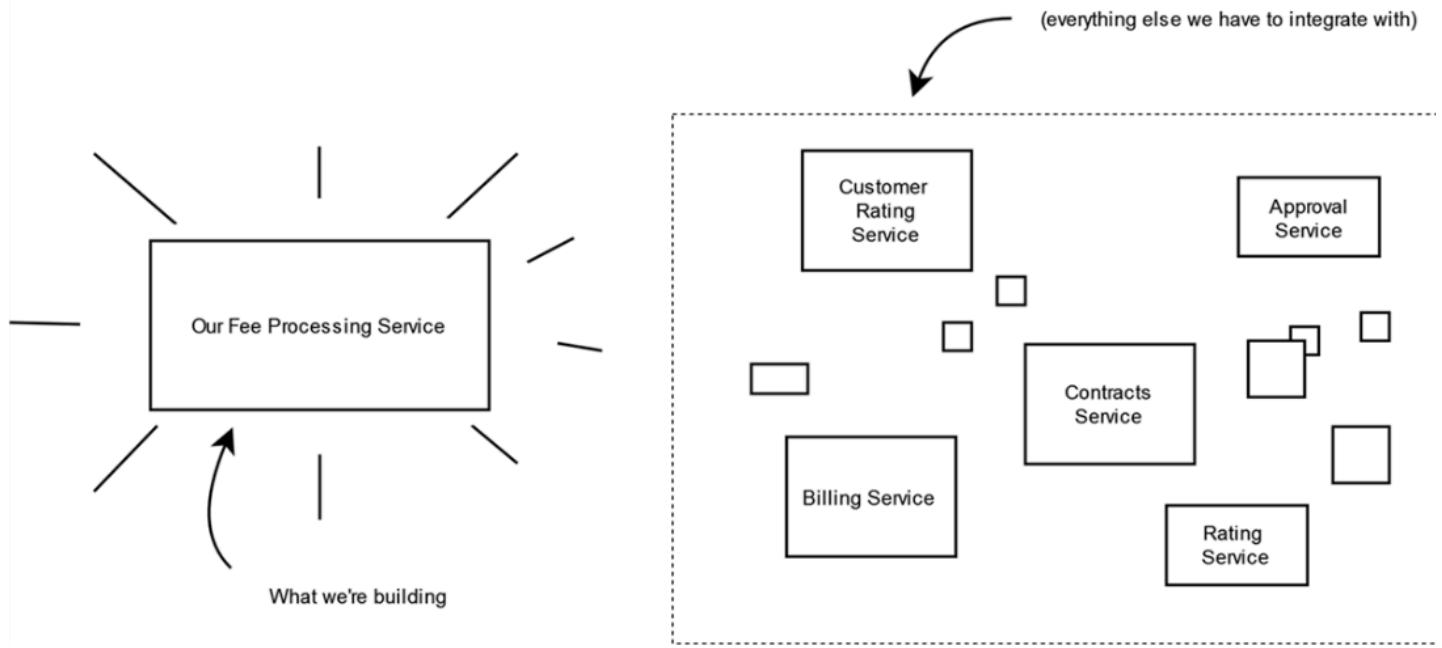


Figure 5.1 The service-oriented landscape in which our application lives

Our service is a small cog in a large sea of other (not so micro) services. We own who should be charged, how much, *why*, and *when*.

It's a lot to take in, and it's going to take a bit of "setting the stage," but this complexity lets us see how data-orientation fits into real world systems. Despite this jump in complexity, we'll still stick with the same strategies we've used so far: focus on the data.

5.1.1 Invoicing (briefly)

Invoices are documents that say how much customers owe us for the services they've used.

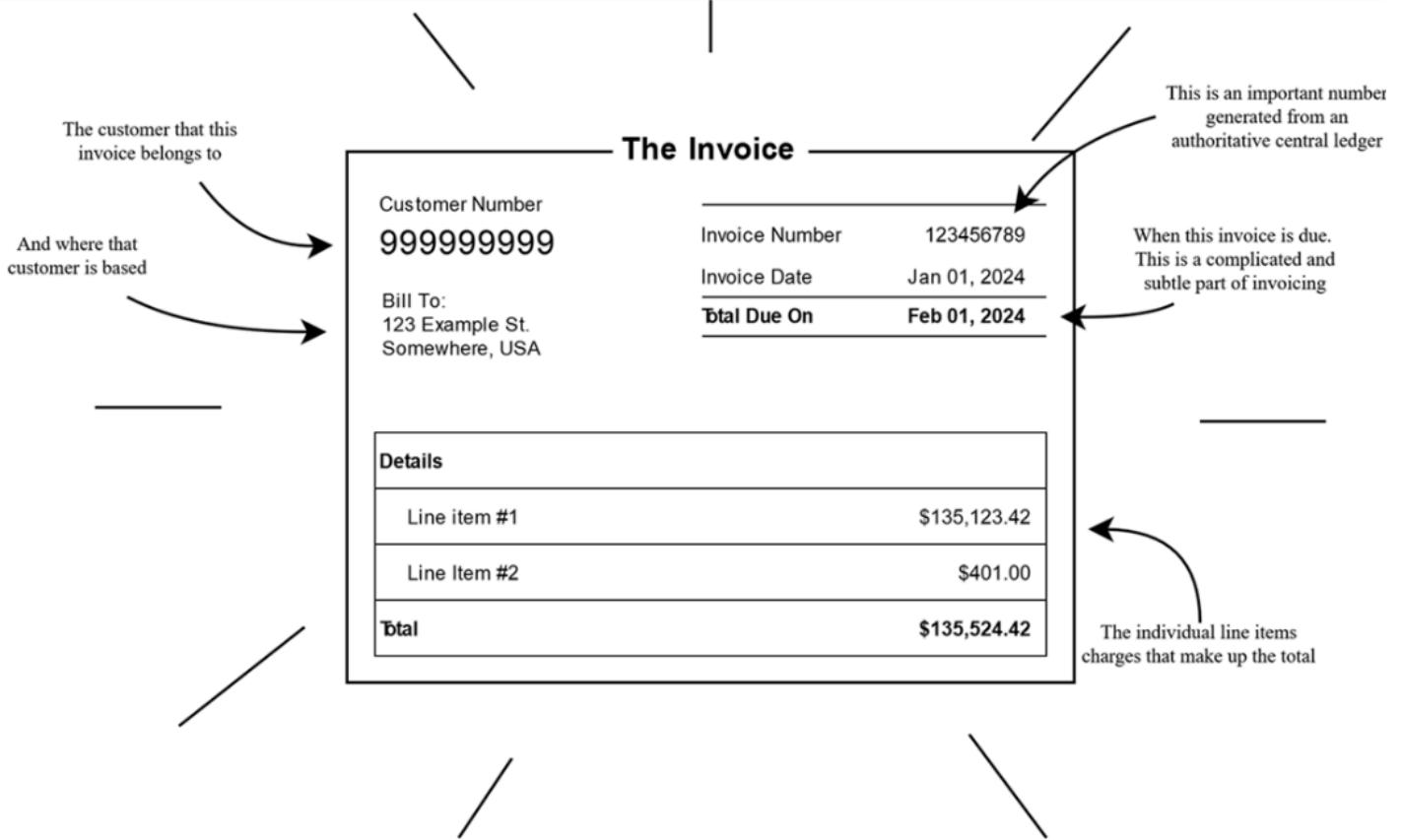


Figure 5.2 The invoice! They're what our service exists to process!

They follow an extremely simple lifecycle. They start out as "open" when issued to the customer, then become "closed" once paid.

Invoices have two important dates: the *invoice date* (when we issued it) and the *due date* (when they have to pay). Due dates vary depending on the customer's *payment terms*.

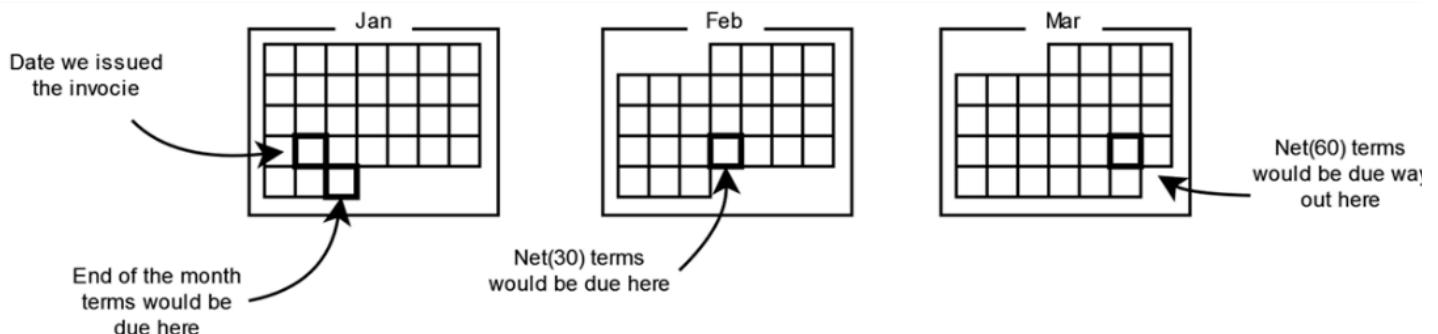


Figure 5.3 How payment terms affect invoice due date

These due dates are also *squishy*. When deciding if an invoice is late, we don't just go off the date as written on the invoice, we factor in the customer's *rating*. If they're a good customer, we add a generous grace period to their due date. If not, they get charged fees much sooner.

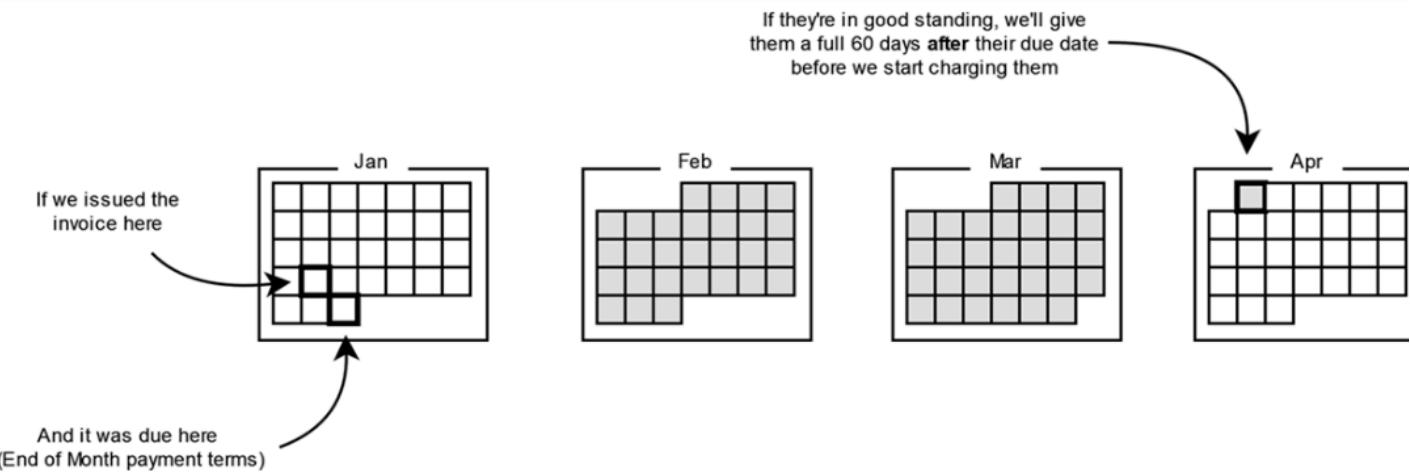


Figure 5.4 grace periods determine when we consider an invoice "past due"

Last thing to know: invoice numbers are special and have to be handled by an external system (centralized sequencing helps prevent fraud). The invoices we make in our system aren't "real" until this central billing service has allocated a number and delivered the invoice to the customer. Our service is just a middle man.

This central billing service segues us into the hellscape in which we'll implement this feature: the legacy system.

5.1.2 The Legacy System!

In modern "service oriented" fashion, information and functionality will be fragmented across (often arbitrary feeling) web services. We have to interact with all of these to complete our feature.

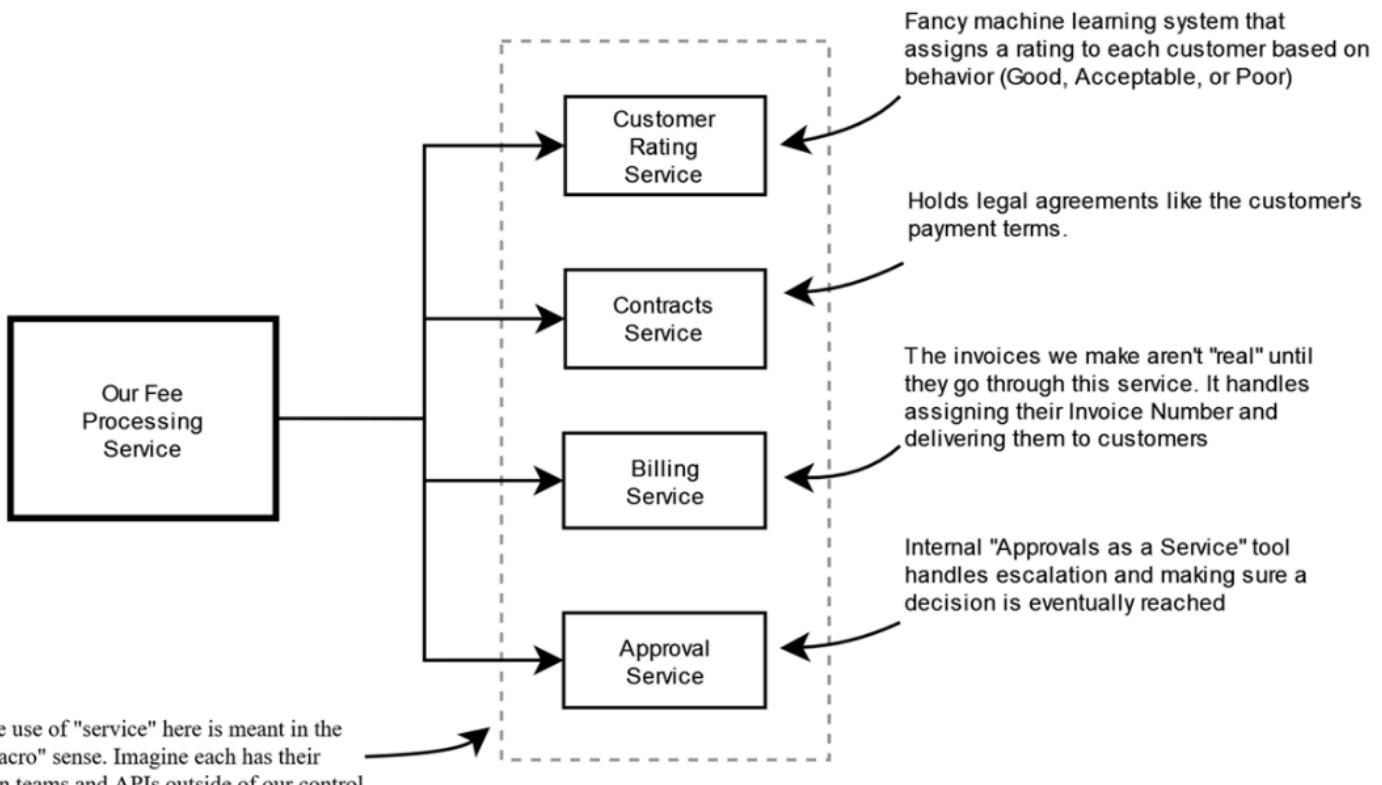


Figure 5.5 Services for every little thing

To keep things simple(ish), we'll ignore HTTP wrappers and failure modes. They'll return the relevant data and that's it. Assume all are blocking and synchronous.

Listing 5.1 Quick sketch of the APIs and data types for the service integrations

```
interface RatingsAPI {                                     #A
    enum CustomerRating {GOOD, ACCEPTABLE, POOR}
    CustomerRating getRating(String customerId);
}

interface ContractsAPI {                               #A
    enum PaymentTerms {
        NET_30, NET_60,
        END_OF_MONTH, DUE_ON_RECEIPT}
    PaymentTerms getPaymentTerms(String customerId);
}

interface ApprovalsAPI {
    enum Status {PENDING, APPROVED, DENIED}
    record Approval(String id, Status status){}
    Approval createApproval(CreateApprovalRequest request); #B
    Optional<Approval> getApproval(String approvalId);      #B
}

interface billingAPI {
    enum Status {ACCEPTED, REJECTED}
    record SubmitInvoiceRequest(...) {}
    record BillingResponse(
        Status status,
        String invoiceId,
        String error
    ){}
    BillingResponse submit(SubmitInvoiceRequest request);   #C
}
```

#A These are the easiest. They're read only.

#B Approvals are more complicated. We have to manage two APIs (read and write).

#C This is the most important and dangerous API in our codebase. Invoices we send through here go out to the customer with no "undo" button

Are we done with the setup? Can we get to the rest of the book now? No! It gets worse! To show how data-oriented programming works in real life, we need all of the pre-existing stuff that makes programming hard. We need databases, and entities, and ORMs, and most terrifying of all: *prior decisions*. Ugh.

5.1.3 A bunch of prior decisions designed around "free"

Keeping with the theme of “design choices already made,” the existing application will be built around a core set of identity objects. Popular frameworks encourage this style of modeling – if not directly, then as a consequence of making it “easy” to do so. A few magical annotations can turn any identity object into a database model. A few more can send that same object out to the world as JSON. Once you’ve got that power at your fingertips, it can be tough to resist. Thus, entire applications end up built around just a few objects. *The Invoice*. *The Customer*.

So, someone (not us) picked out “the nouns.”

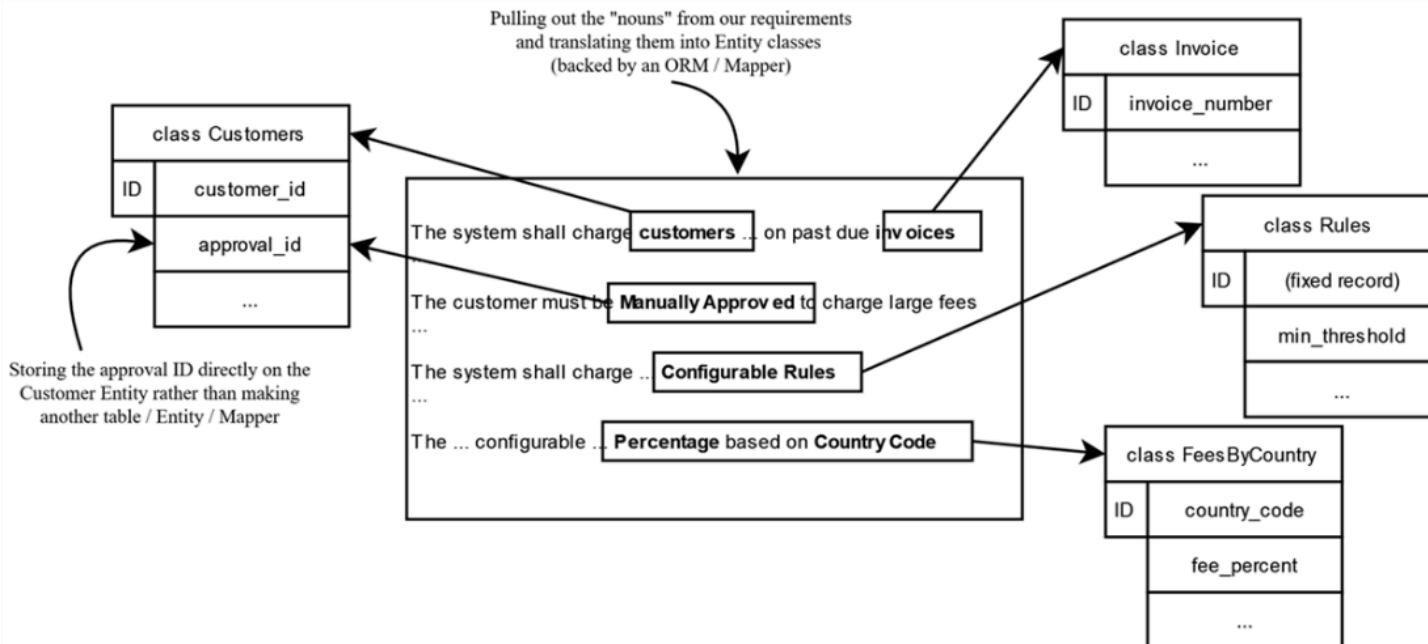


Figure 5.6 Pulling Entities Objects / Tables from the requirements

These “nouns” get turned into Identity Objects, and some magical annotation library (we’re purposefully ignoring the specifics here) handles mapping them into/out of a database.

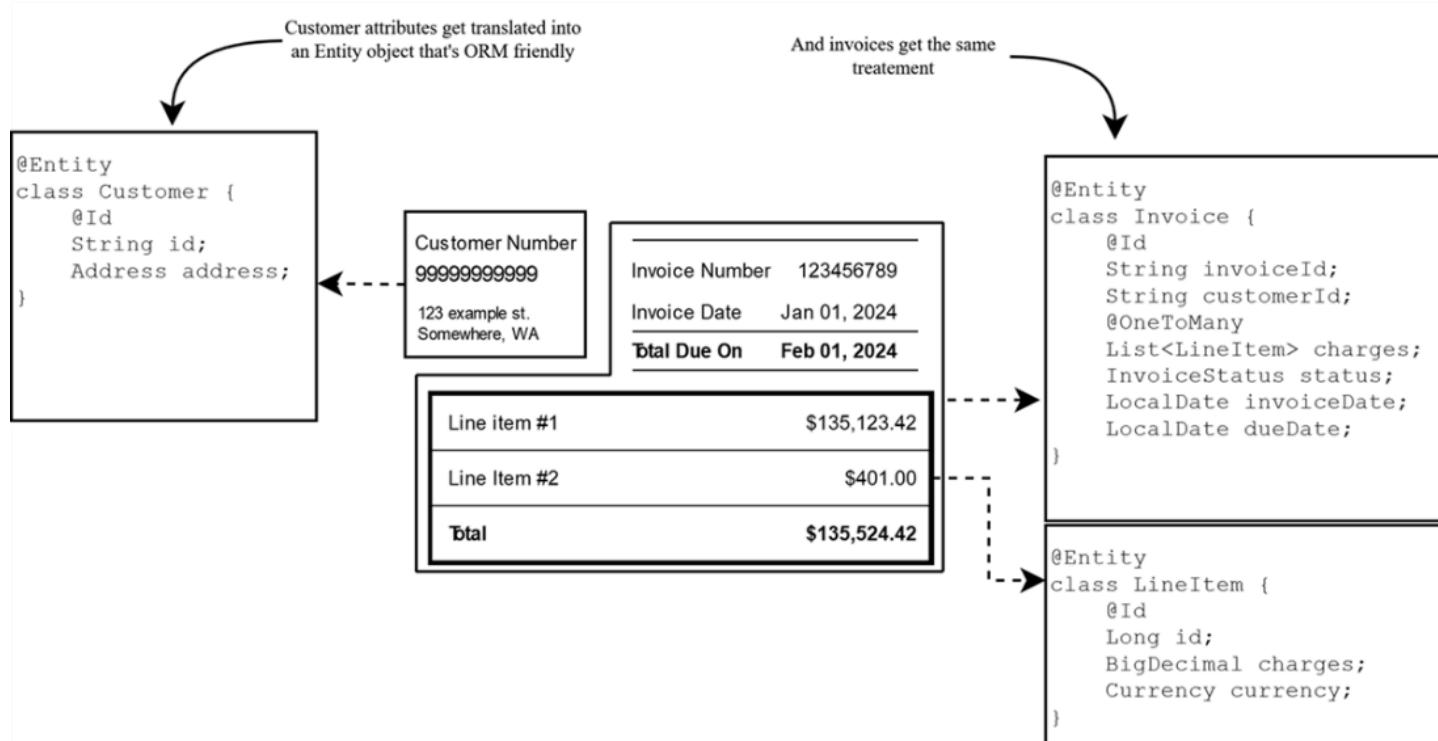


Figure 5.7 turning the Invoice info into (ORM backed) Entities

Hard modeling problems (“should a Late Fee Invoice be modeled differently from a ‘normal’ Invoice?”) will be handled in the way that ORMs make easiest: ignoring them. Instead, we’ll add enums and nullable fields to the “canonical” representations as needed.

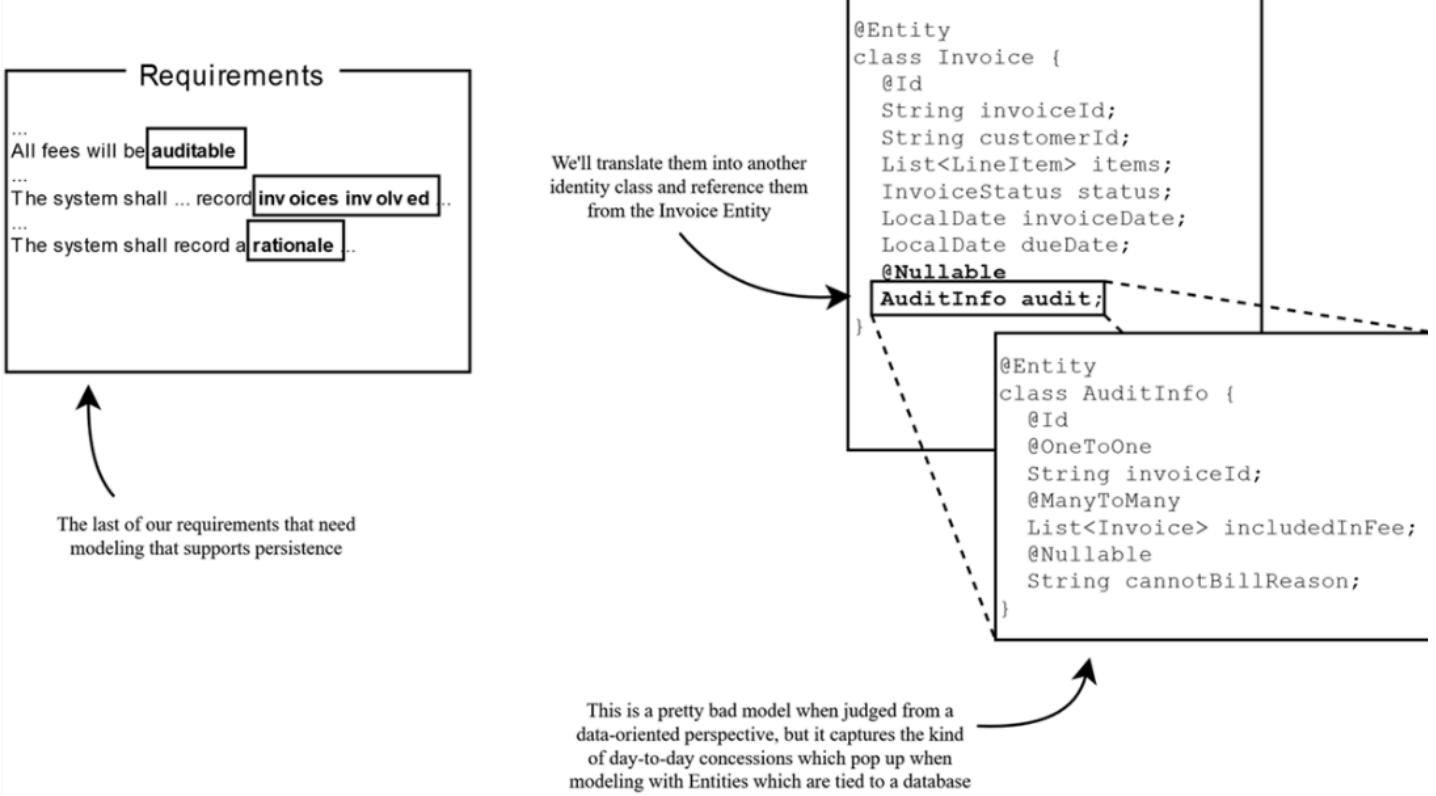


Figure 5.8 the small concessions we make when our representation is tied to persistence

This yields the core set of identity classes around which we'll have to build the feature.

Listing 5.2 The core Entities we're starting with

```
enum InvoiceType {LATEFEE, STANDARD};    #A
enum InvoiceStatus {OPEN, CLOSED}

@Entity                      #B
class Invoice {
    @ID                      #B
    String invoiceId;
    String customerId;
    @ManyToOne                 #B
    List<LineItem> lineItems;
    InvoiceStatus status;
    LocalDate invoiceDate;
    LocalDate dueDate;
    InvoiceType type;
    @Nullable
    AuditInfo auditInfo;      #C
}
@Entity
class LineItem {
    @Id
    String id;
    String description;
    BigDecimal charges;        #D
    Currency currency;         #D
}
@Entity
class AuditInfo {
    @Id
    @OneToOne
    String invoiceId;
    @ManyToMany
    List<Invoice> includedInFee;
    @Nullable
    String cannotBillReason;
}
@Entity                      #E
class Customer {             #E
    @ID
    String customerId;
    Address address;
    @Nullable
    String approvalId;        #F
}
```

#A This enum tells us what kind of Invoice the Invoice Entity is representing

#B (All of the annotations are made up. Assume they map these classes into some database somehow)

#C This will be populated when InvoiceType is LATEFEE, otherwise null

#D Built in Java types are used on these core entities rather than more data-oriented ones. ORMs usually make customizing types tedious enough that most won't bother.

#E The canonical Customer entity

#F Approval ID is tracked here for "ease"

Each Entity will each have a Repository (or “Data Access Object”, depending on your lingo) generated. Assume standard CRUD style methods. All of that will get crammed into another class (often called a “service”) where we’ll write our business logic.

Listing 5.3 Where most features begin

```
class LateFeeChargingService {          #A
    private RatingsAPI ratingsApi;       #B
    private ContractsAPI contractsApi;   #B
    private ApprovalsAPI approvalsApit;  #B
    private BillingAPI billingApi;       #B
    CustomerRepo customerRepo;          #C
    InvoiceRepo invoiceRepo;            #C
    RulesRepo rulesRepo;                #C
    FeesRepo feesRepo;                 #C

    public void processLateFees() {        #D
        // So we begin!
    }
}
```

#A The class where we write our business logic. We call it a “service” because that’s usually what most frameworks have us call these things.

#B The external services on which we depend

#C The repositories (Data Access Objects) for all of our Entities

#D Now we’re ready to start!

5.2 Modeling domain behaviors around existing code

Listing 5.3 is the shell around which a lot of our features begin. We have a fleet of repositories and services at the ready. We can conjure up any Entity we need. There’s so much stuff already there that it can feel like the “design” part is done. All that’s left for us is assembling the pieces.

A first stab at that assembly might look like this.

Listing 5.4 Building on top of what's already there

```
public void processLatefees() {
    LocalDate today = LocalDate.now();

    Rules config = rulesRepo.loadDefaults();                                #A
    for (Customer customer : customerRepo.findAll()) {                      #A
        BigDecimal feePercentage = feesRepo.get(
            customer.address().country()                                         #A
        );
        List<Invoice> pastDueInvoices = getPastDueInvoices(customer);      #B
        BigDecimal totalPastDue = getTotalPastDue(pastDueInvoices);         #C
        BigDecimal latefee = totalPastDue.multiply(feePercentage);           #C

        Invoice latefeeInvoice = new Invoice();                                #D
        latefeeInvoice.setInvoiceType(InvoiceType.LATEFEE);
        latefeeInvoice.setCustomerId(customer.getId());
        latefeeInvoice.setInvoiceDate(today);
        latefeeInvoice.setDueDate(this.figureOutDueDate());                   #E
        latefeeInvoice.setLineItems(List.of(new LineItem(
            null,
            "Late Fee",
            latefee,
            Currency.getInstance("USD")))
    );
    latefeeInvoice.setAuditInfo(
        new AuditInfo(null, pastDueInvoices, null));

    if (latefee.compareTo(config.minimumFeeThreshold()) <= 0) {          #F
        latefeeInvoice.auditInfo().setReason("too low to charge!");
        invoiceRepo.save(latefeeInvoice);
    } else {
        if (latefee.compareTo(config.maximumFeeThreshold()) > 0) {
            if (customer.getApprovalId().isEmpty()) {
                this.requestReview(latefeeInvoice, customer);
                latefeeInvoice.getAuditInfo()
                    .setReason("Above default threshold");
            } else {
                ApprovalStatus status = getApprovalStatus(customer);
                if (status.equals(ApprovalStatus.PENDING)) {
                    latefeeInvoice.getAuditInfo()
                        .setReason("Pending decision");
                } else if (status.equals(ApprovalStatus.DENIED)) {
                    latefeeInvoice.getAuditInfo()
                        .setReason("Exempt from large fees");
                }
            }
            invoiceRepo.save(latefeeInvoice);      #G
            continue;                           #G
        }
    #H
    invoiceRepo.save(latefeeInvoice);
    this.submitBill(latefeeInvoice);
}
}

List<Invoice> getPastDueInvoices(Customer customer) {    #I
    CustomerRating rating = this.ratingApi.getRating(customer.id());
    return invoiceRepo.findInvoices(customer.id())
        .stream()
```

```

.filter(invoice -> {
    if (rating.equals(CustomerRating.GOOD)) {
        return invoice.getDueDate()
            .plusDays(60).isBefore(LocalDate.now());
    }
    else if (rating.equals(CustomerRating.ACCEPTABLE)) {
        return invoice.getDueDate()
            .plusDays(30).isBefore(LocalDate.now());
    } else {
        return invoice.getDueDate()
            .with(lastDayOfMonth()).isBefore(LocalDate.now());
    }
})
.toList();
}
// (Other helper methods skipped for brevity)
}

```

#A We start out just grabbing a bunch of the entities we need

#B Implemented below

#C Some logic for computing the fee total

#D Now we begin building up our result – the new Late Fee

#E (We'll dig into these methods later)

#F Now the complexity quickly ramps up. What do all of these conditionals do?

#G Can you tie this branch back to a specific requirement? Which one(s)?

#H This branch silently handles a whole suite of requirements. Wrapped up in what's not said are some of the most important invariants in our system: who gets charged money

#I This "helper" method does a lot of the heavy lifting.

We could go line by line with a red pen, but instead let's focus on this: notice how little conscious design there is. The entire feature is assembled out of the debris of *other people's* design choices. Its shape has been dictated by services we don't control, fixed entity classes we didn't pick, and access patterns that were pre-determined.

Listing 5.3 will surely look different from how you might have implemented it, but I bet that it feels familiar -- like code you could have written. We've all written code like this. It's stream of consciousness and mechanical. All the pieces were laid out ahead of time. We just put them together.

A single feature written like this isn't bad, but a code base written like this is a nightmare. What's missing is *why* this code is here. What does this mess of nested if statements and services calls do? How do they relate to the requirements? What *are* the requirements? They're in there somewhere, but you have to read between the lines to see them. None of what we're trying to do is explicitly stated.

This isn't a matter of style or abstractions. Services and entities designed around some *other* use case will seldom fit *ours*. Yet, we try to make them. Building new features on top of what's already there (just *because* it's already there) dooms us to the semantic woes we spent the last several chapters exploring. Their imprecise representation creates codebases which lie to us.

Listing 5.5 Code that's filled with lies

```
public void submitBill(Invoice invoice) { #A  
    if (invoice.invoiceType().equals(LATEFEE)) #B  
        && invoice.getInvoiceId() == null #C  
        && invoice.getAuditInfo() != null #D  
        && Strings.isNullOrEmpty( #E  
            invoice.getAuditInfo().getReason())) { #E  
  
    Response response = this.billingService.submit()  
    // ...  
  
} else {  
    throw new IllegalArgumentException(  
        "Whoa! You're about to charge something you shouldn't!"  
    )  
}  
}
```

#A There's no semantic delineation between the kinds of Invoices in our system, so the interface we expose lies about what's allowed.

#B The first thing we have to do inside of this method is mount our defenses against the incoming data

#C We don't bill any Invoice! We only bill Late Fees, and that are in a certain state, and that haven't already been billed. Anything else is an error.

#D These checks are doomed to be cryptic and confusing – they rely on you knowing when particular fields should be set and what it means when they're not. We do a null check here because we shouldn't have an InvoiceId yet -- we only get that after we bill.

#E Ditto here. Try connecting this null check back to a particular requirement!

Listing 5.5 holds the most dangerous method in our entire project. It's the one that charges people money. There's no "undo" button. It should be bullet proof against misuse.

But it isn't. Instead, it starts off with a lie. It advertises to the world that it can charge "any Invoice", but secretly devotes its implementation to defending against that mistruth. It should only process the *Late Fee* invoices that *our* system generates (and even of those, only a small subset). Anything else is a horrible error.

Worst of all is that these "defenses" we've laid have no clear semantic relation to any business rule. They're mired in the details of how our entities map into the database. We don't assert "we shouldn't have already charged this," we assert that certain fields are null. It's left to the reader to figure out what those null checks mean. Those three lines of cryptic defensive coding in Listing 5.5 are all stands between the customer and a torrent of incorrect bills.

Speaking of which...

This scenario is from one of my most expensive programming mistakes (so far!). It stemmed from this exact kind of misleading code. There was a method in our codebase that sent bills to customers (for *lots* of money). The design of this method allowed any invoice to be passed to it (just like Listing 5.5), so, inevitably, the *wrong* kind of invoice ended being passed to it. These "wrong" invoices got through our runtime defenses in the usual way: a pedestrian misunderstanding about what the code *should* do.

These mistakes don't need to happen.

We can apply the same skills we learned for modeling *data* to modeling behaviors that operate on data. We can make the behaviors in our domain similarly expressive, self-describing, and self-enforcing. Though our modeling, we can make the incorrect use of powerful methods impossible.

5.3 Modeling domain behaviors around data

When we think about behaviors in a system, it usually evokes images of “doing” or “actions” – something fleeting and ephemeral that causes change somewhere in the world. The void method. Data-orientation takes a more “mechanical” view of the world. Behaviors are discrete. They change one state into another. Inputs produce outputs.

This is a pretty normal way of thinking for many parts of our programs. `toString()` is a good example. It’s a very mechanical operation. It takes in an Object on one side and produces a String on the other. The data-oriented view is to keep up this mechanical “inputs and outputs” vantage point even as the operations and state changes become more complex. This includes even ephemeral actions like calling an API. We can model that call as two discrete states: the one before the call, and the one after.

What are these states? They’re data we have to invent. Somewhere in our requirements is a “machine” that we can model. Like a factory, raw Invoices come into our “machine” on one side, Late Fees come out the other.

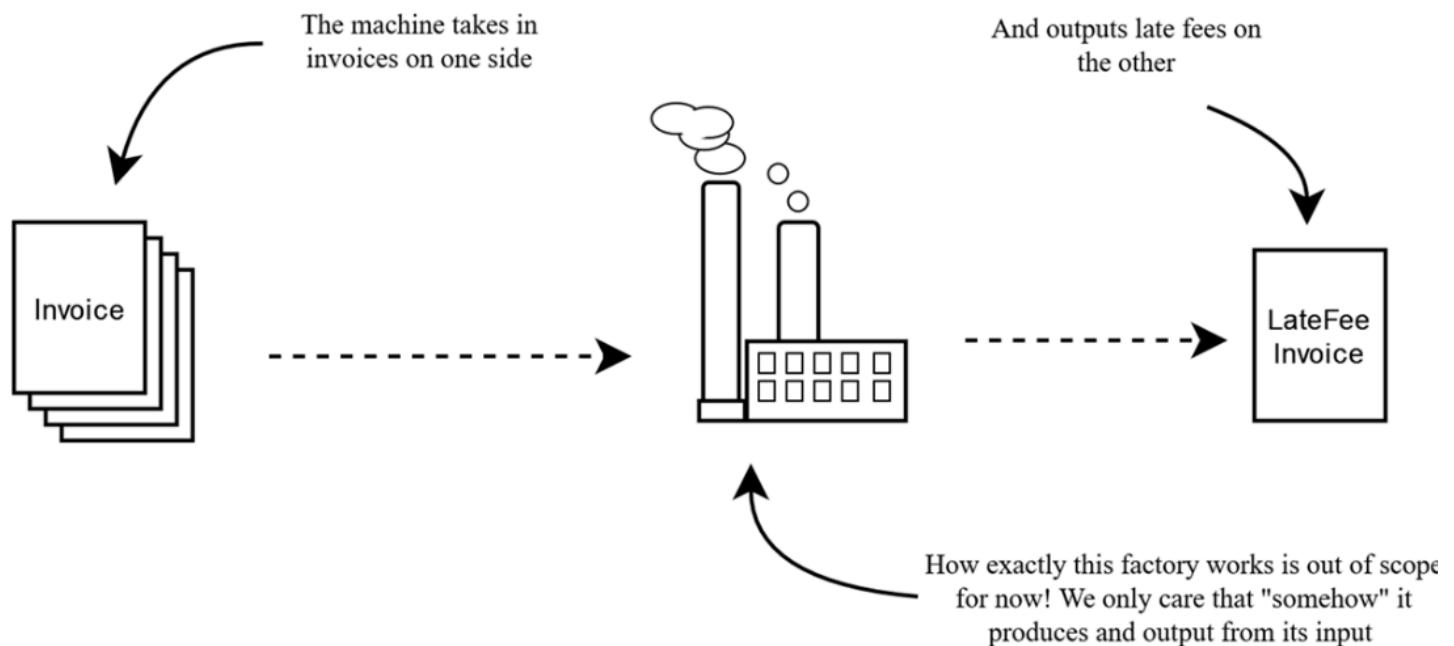


Figure 5.9 Visualizing behaviors as a factory. Data in; Data out.

The design process is figuring out how many of these little “machines” we need and what each one processes.

It’s not that different from the modeling we did in previous chapters. The only big change is that rather than focusing on a piece of data in isolation, we focus on how pairs of data transform from one to another.

Also, like before, we'll ignore the implementation details for now. We're going to invent a lot of new data types as we go, but only in *name*. We'll ignore what's inside of them for now. The goal is to stay "above" the implementation details so we can stay loose and flexible while we're designing. We can conjure up a new data type out of thin air, and then scratch it out just as easily if we realize it's no good.

5.3.1 Telling a story with data

We know the input to our system. The Invoice is already made for us. What we need is the output. So, let's start there. We can give ourselves what we know is already missing: a data type called `LateFee`. This type is the whole reason our service exists.

Our "factory" takes a List of Invoices as input and produces a Late Fee as output, which will eventually cause a new invoice to be added to the state of the system.

We'll represent this "jump" from one data type to another by drawing an arrow (`->`) between them. What does that arrow do? How is it implemented? It doesn't matter! We'll figure it out later. The important part is the relationship that this magical arrow creates between the input data and the output data. When we put them next to each other, they should tell us a story.

Listing 5.6 A transform from one data type to another

```
List<Invoice> -> LateFeeInvoice
```

We read this as saying that we start with a list of invoices, and then use those to produce a new late fee invoice. Just like with the arrow (`->`) itself, we won't worry about what these data types are. Is a `LateFeeInvoice` related to an `Invoice`? Part of a hierarchy or sealed family of classes? It doesn't matter at this point – they're *semantically* different, and that's what we're trying to capture. We'll deal with how they're *implemented* later.

These transformations between data types form the basic building blocks our story. They're how we communicate the core ideas in our domain. So, each time we add a new pair of data types, we should pause to see what kind of story they're telling, and if that story is the right one.

If you imagine describing this feature to a coworker, you'd probably notice that the current story, "Invoices are used to make late fees," misses a lot of subtlety. It's not *any* invoice, it's specifically *past due invoices* that produce late fees. The state of the `Invoice` is important. That's why we're charging fees in the first place.

We can clarify the story by adding another step with a more descriptive data type.

Listing 5.7 Refining how late fees get produced in our system

```
List<Invoice>          #A  
-> List<PastDue>    #B  
-> LateFeeInvoice    #C
```

#A We start with a list of invoices

#B then pare those down to Past Due invoices

#C and use those to build a single Late Fee Invoice

What's cool about just playing around with data types like this is how quickly the story comes to life. We're able to communicate the core ideas in the requirements without writing a single line of implementation code.

Let's keep fleshing out this story.

One of the core subtleties in our system is that the fee invoice we make isn't "real" until we submit it to the billing system. What we're really doing at this stage is creating a description of invoice that we want to have *later*. We can think of it as special part of the lifecycle. It starts out in our system as a `Draft`, then reaches its official invoice state of `Open` once it's sent through the billing system.

Lifecycles like this pop up everywhere in programming. They're not part of the "real" domain (Invoices officially only have two lifecycle states: Opened or Closed), but exist as a consequence of the software systems in which they live. It's important to capture them. Let's refactor our pipeline to make that lifecycle state explicit. (Note here that "refactor" just means slotting in a new type. This ease of shuffling things around is why we stay up above the implementation details. It massively lowers the cost of iteration while we design.)

Listing 5.8 Introducing the draft lifecycle

```
List<Invoice>
-> List<PastDue>
-> LateFeeDraft      #A
-> LateFeeInvoice
```

#A Capturing the fact that in our domain, past due invoices are used to create late fee drafts

This is beginning to capture something interesting. If you mentally replaced each arrow with a phrase like "and then we use that to make...", you'd end up with something that really does begin to read like a simple story about our requirements.

Listing 5.9 In plain English

```
(we start with) List<Invoice>
(and then we use that to make) List<PastDue>
(and then we use that to make a) LateFeeDraft
(and then we use that to make a) LateFeeInvoice
```

Under this light, you might notice that our "story" is still missing a beat. It says Drafts become Invoices, but, in reality, only *some* drafts will go on to become LateFeeInvoices *some of the time*. The "why" and "when" are the most complicated parts of our requirements -- also the most complicated part of our original implementation.

Listing 5.10 The original implementation

```
if (latefee.compareTo(config.minimumFeeThreshold()) <= 0) {  
    // [logic omitted]  
} else {  
    if (latefee.compareTo(config.maximumFeeThreshold()) > 0) {  
        if (customer.getApprovalId().isEmpty()) {  
            // [logic omitted]  
        } else {  
            if (status.equals(ApprovalStatus.PENDING)) {  
                // [logic omitted]  
            } else if (status.equals(ApprovalStatus.DENIED)) {  
                // [logic omitted]  
            }  
            // [logic omitted]  
        }  
    }  
    // [logic omitted]  
}  
// [logic omitted]
```

Inside of that wall of nested if/else conditions sits a really simple *idea* in our domain: deciding what to do with the draft. It's either billable, or needs a manual approval, or below the threshold where we won't bother the customer. It's just hard to see those three discrete decisions because the code is forced to juggle so many other concerns.

But these decisions are the important stuff. It's what we want the people who read our code to understand. To do that, we have to lift it out of the implementation and capture it as data.

5.3.2 Represent runtime decisions as compile time algebraic data types

Any decision we make using if/else statements at runtime can be expressed algebraically at compile time as data.

Listing 5.11 Runtime if/else as Compile time data

```
void example() {  
    #A  
    if /* option #1 */ {...} #A  
    elif /* option #1 */ {...} #A  
    else {...}  
}  
  
sealed interface Decision {  
    #B  
    record Option1() implements Decision {} #B  
    record Option2() implements Decision {} #B  
    record Option3() implements Decision {} #B  
}
```

#A Decisions we make at runtime using if/else
#B can be represented at compile time using sealed (sum) types

This modeling approach has powerful implications because it decouples where we make a decision from where we act on it. Conditional expressions are almost always paired with their associated action.

Listing 5.12 A decision and its actions

```
if (someCondition) {    #A
    doSomeAction();      #A
} else {
    doSomeOtherAction();
}
```

#A The decision and its action are executed together

However, if we return the decision itself as data, we get to defer the *action* part till later (if at all!). We can build custom *interpreters* for those decisions and decide how and where we want to handle them.

Listing 5.13 Interpreting the decisions

```
String doSomethingWithTheDecision(Decision decision) {
    return switch(decision) {
        case Option1 op -> "I do something with option1";
        case Option2 op -> "I do something with option2";
        case Option3 op -> "I do something with option3";
    }
}
```

This buys us a ton of modeling flexibility. Most importantly, it allows us to lift an otherwise hidden part of our domain up to the top where everyone can see it.

Listing 5.14 Capturing the decisions our application can make

```
List<Invoice>
-> List<PastDue>
-> LateFeeDraft
-> (BillableFee OR NotBillable OR NeedsApproval)  #A
-> LateFeeInvoice  #B
```

#A These options capture everything our system can do with a draft

#B Now that we can see the choices, it's clear that this needs some more thought

And putting it at the top has some immediate effects on the “story” our data types are creating. It forced a new level of precision into our modeling. With the branching on full display, it makes it really clear that our initial “end point” was wrong -- or at least *imprecise*. Drafts will end up in one of three buckets, and only one of them will go on to become a LateFeeInvoice.

Listing 5.15 Clarifying which of the three decisions can lead to a Late Fee

```
List<Invoice>
-> List<PastDue>
-> LateFeeDraft
-> (BillableFee      -> LateFeeInvoice  #A
     OR NotBillable   -> ???          #B
     OR NeedsApproval -> ???)         #B
```

#A Now we're really becoming precise

#B The huge benefit here is that we're forced to acknowledge that these cases exist, and we need to do something about them

But lurking in the background are more life cycle states. If you carefully study the billing API from way back in Listing 5.1, you'll notice that we're not guaranteed to get

back an invoice. The service can decide that our late fee isn't billable. So, our story doesn't conclude with the production of a singular late fee. It ends with a late fee in a particular state – one that's either billed, or not billed.

Listing 5.16 Capturing the lifecycle phases of a Late Fee

```
List<Invoice>
-> List<PastDue>
-> LateFeeDraft
-> BillableFee      -> (BilledLateFee OR RejectedLateFee) #A
  OR NotBillable    -> ???
  OR NeedsApproval  -> ???
```

#A Capturing the final lifecycle status of a Late Fee we've submitted to the billing system

That's the hard part done!

The rest of the process follows the same pattern. We match invented inputs to invented outputs, and keep refining the story until we've got a complete map of how data flows through the program.

Listing 5.17 Handling the remaining branches

```
List<Invoice>
-> List<PastDue>
-> LateFeeDraft
-> (BillableFee      -> (BilledLateFee OR RejectedLateFee)
  OR NeedsApproval  -> ApprovalStarted #A
  OR NotBillable    -> RejectedLateFee #B)
```

#A The nice thing about really descriptive types is that they largely push themselves in the right direction. What kind of output will something that takes a NeedsApproval as input produce? Something to do with an Approval!

#B Ditto here. If it's not billable, it ends up in our terminal "Rejected" late fee state

Take a minute to notice how different this feels as a starting point. You can see our requirements directly in the data types. We don't even need the implementation. The data types tell us a story.

Playing around with data like this is a powerful design tool. It allows us to refactor and refine this pipeline endlessly at no cost because we haven't committed to anything -- we don't even know what these data types will ultimately look like. This process takes many pages to describe in a book, but only a few minutes of effort in real life. Designing behaviors this way lets you rapidly explore different approaches and "step" sizes. It lets you build up a high-level story of how the data transforms as it moves around your system.

So, when are we done?

That answer will vary from person to person. The endpoint I shoot for is to keep refining until I end up with "baby code." I want it to read like a set of instructions written for a human rather than a computer. "Done" is when I can see all of the major requirements as a narrative.

Listing 5.18 A narrative designed for human consumption

```
// step 1: collect the past due
List<Invoice> -> List<PastDue>

// Step 2: use those to build the draft
List<PastDue> -> LateFeeDraft

// Step 3: decide what to do with the draft
LateFeeDraft -> (BillableFee OR NotBillable OR NeedsApproval)

// then depending on what we decided, either:
// Step 4.1: submit billable items
BillableFee -> (BilledLateFee OR RejectedLateFee)

// OR Step 4.2: Start the approval process for those that need it
NeedsApproval -> ApprovalStarted

// OR Step 4.3: keep the non billable data for posterity
NotBillable -> RejectedLateFee
```

Once you're happy with it, you can take each step and turn it into a method. Inputs feed outputs. Outputs feed the next set of inputs. These become a high-level guide to our implementation. These transformations together make up what our behavior *is*.

Listing 5.19 Converting the design into methods

```
List<PastDue> collectPastDue(List<Invoice> invoices) {...}
LatefeeDraft buildTheDraft(List<PastDue> invoices) {...}
ReviewedDraft assesTheDraft(LatefeeDraft invoice) {...}
BillingResult submitBill(LatefeeDraft draft){...}      #A
ApprovalStarted startApproval(NeedsApproval needsApproval) {...} #A
```

#A Even though these enact external change in the world, we still design them as though they're machines that take inputs and produce outputs. The result of their action is new information. New data.

Of course, these won't work yet – they're just a rough sketch. They're missing all the other stuff in our domain that comes from external service calls and repositories. They're here as a high-level guide for our implementation.

5.4 Fleshying out the representation

We have to decide how we'll represent the data we invented in our sketch. This puts us back in familiar territory of modeling individual "nouns." However, the real world brings challenges to that modeling process that weren't present in previous chapters.

In addition to making sure the data types are self-describing and semantically clear, we also have to ensure that their semantics fit the *behaviors* we've designed. Each time we tweak our data's representation, we have to take a step back and check how those choices have rippled outwards through the system. Good representations remove bad states and potential bugs; weak representations create them.

Additionally, we have to pay super close attention to the ergonomics of the data types we're creating. Clear semantics are great, but if getting those semantics into our code is painful or awkward, most developers won't put up with the annoyance. We have to ensure that clarity doesn't get in the way of our quality of life.

These challenges mean that we won't always get it right on our first try. Design is an iterative and messy process. The first solution we come up with is just that: the *first* one. There will be many. Design is the exploration of these possible solutions. The more time we spend exploring, the better the odds we'll find the right tradeoffs.

So, we have some complicated modeling decisions to make. We introduced several new data types in section 2.3. Our representation of these types and how we relate them together will have a massive influence on how our code feels and the kinds of invariants it can enforce.

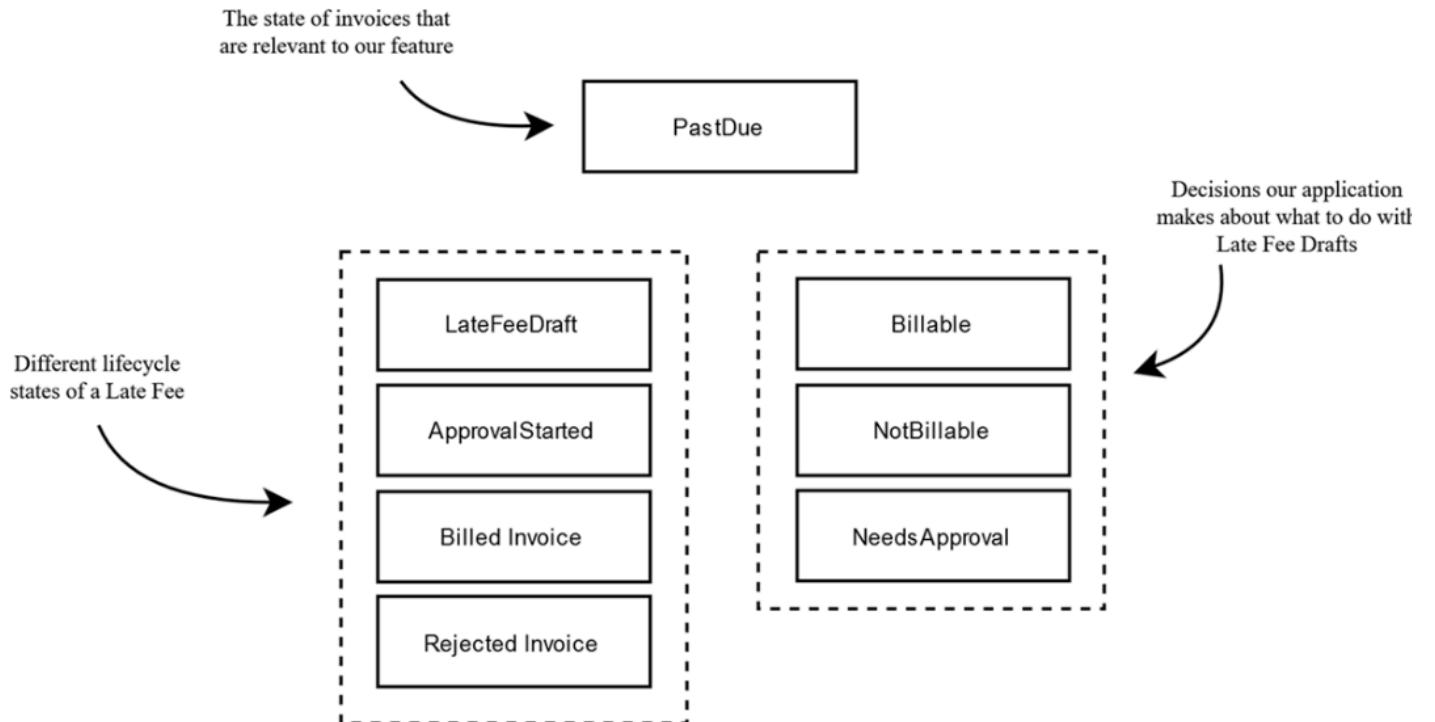


Figure 5.10 The new data types that need design

5.4.1 Creating wrapper types and being OK with comingled Identities

We'll start simple with the `PastDue` data type. It stands out from the rest because it's purely informational. It doesn't track any unique state or have any novel attributes of its own. Past Due represents something we know about the state of an `Invoice` entity.

One way of capturing this "stuff we know" is by creating simple wrappers. Records make this really lightweight to do.

Listing 5.20 Creating the `PastDue` type

```
record PastDue(Invoice invoice){} #A
```

#A Creating a lightweight wrapper type around the `Invoice` entity

Once we have a new data type, we can ask our usual questions about semantics. And that leads us to one of the unfortunate realities of building on top of what's already there: `Invoice` is a mutable identity class, not a value. So, even though we want to design and program with data, we're getting dragged back down into the world of identities and change.

Listing 5.21 A closer look at that same model

```
record PastDue (          #A
    Invoice invoice      #B
) {}
```

#A We want this to be an immutable piece of data

#B but we're building it on top of a mutable identity object (the Invoice entity)

In situations like this where we're bridging the gap between the way we used to do things (mutable entities) and how we're starting to do things (programming with data), there's going to be some unavoidable mismatching.

And that's ok!

Data-orientation is not about purity. What we want is clarity. All of our modeling is in pursuit of that goal. Immutable data makes that goal a bit easier to reach, but it can still be achieved even if we bend or break "the rules" from time to time.

Of course, we could also "fix" this problem. We could make an immutable version of Invoice, or model PastDue so that it holds immutable copies of the relevant fields (rather than just wrapping the mutable identity object), but we also don't have to do any of that. We can always redraw those boundaries later. For now, wrapping can be enough. It acts as a quarantine for the Invoice's mutability. In order to modify that Invoice in place, you'd have to ignore the "spirit" of what records are supposed to be and actively abuse their shallow immutability.

5.4.2 Being OK with imperfect semantics

This one-line wrapper has some more problems in store for us. If we ask our familiar modeling question, "what does it mean to be past due?" we'll get a complex answer that depends on different requirements – many of which involve the state of the world and things outside of the scope of an individual invoice.

These are very different kinds of constraints from the ones we explored in previous modeling exercises. Data types like `NonNegativeInt` and `Degree` are self-contained.

Everything we need to enforce their semantics is available right in their constructor.

Listing 5.22 revisiting NonNegativeInt

```
record NonNegativeInt(int value){      #A
    NonNegativeInt {
        if (value < 0) {            #A
            throw new IllegalArgumentException("Nope");
        }
    }
}
```

#A Enforcing semantic integrity is possible because everything its meaning depends only on its inputs

PastDue is a very different kind of data type.

Listing 5.23 The meaning of “past due” is contextual and depends on outside info

```
record PastDue(Invoice invoice) {  
    PastDue {  
        if (invoice.dueDate().isBefore(/* what goes here? */) #A  
            #B  
        }  
    }  
}
```

#A past due relative to what...?

#B (Plus everything else we'd need to check)

In the ideal world, we want data types that guard their own meaning and prevent us from making mistakes, but PastDue cannot do that as designed. Something can only be “past due” relative to some other date.

Of course, we could redesign the type to track that state.

Listing 5.24 Tracking more data just to enforce semantic integrity

```
record PastDue (  
    Invoice invoice,  
    LocalDate lateAsOf #A  
) {  
    if (invoice.dueDate().isBefore(lateAsOf)) { #A  
        // and is a standard invoice  
        // and is open  
        // and ...  
    }  
}
```

#A If we make the data type carry additional information, we can use that to enforce semantic integrity during construction.

But should we? It really depends. I'll again stress that the goal of data-orientation is not purity. We want clarity of communication. We want code we can understand. Data types are a means to an end.

So, there's not really a right answer here. It can be useful to take a step back and remember why we added this type to our design in the first place. It's there to give a name to something that'd otherwise be implicit or “forgotten” as soon as we leave the bounds of the method that computed it. A draft that says it's built from *past due* invoices communicates something very different from one that says it's built from *any invoice*.

Listing 5.25 What different input types communicate

```
LateFeeDraft createLateFeeDraft(List<Invoice> invoices) {...} #A  
LateFeeDraft createLateFeeDraft(List<PastDue> invoices) {...} #A
```

#A Only one of these tells the truth

Given that, sticking with a bare bones one-line wrapper type, even with all of its short comings and semantic imperfections, seems like a good option to me. It conveys something useful to the reader, and that's enough to make it valuable.

This is the design process. We're constantly weighing different, often competing, concerns. We have to balance between expressivity, power, correctness, and semantics.

5.4.3 Designing the algebra for Late Fees

The remaining data types pose an interesting modeling challenge in Java. `Draft`, `ApprovalStarted`, `Billed`, and `Rejected` are data types that represent the life cycle of a late fee. What makes it tricky to model is how similar these types are. Each type only differs by one or two attributes. They share everything else.

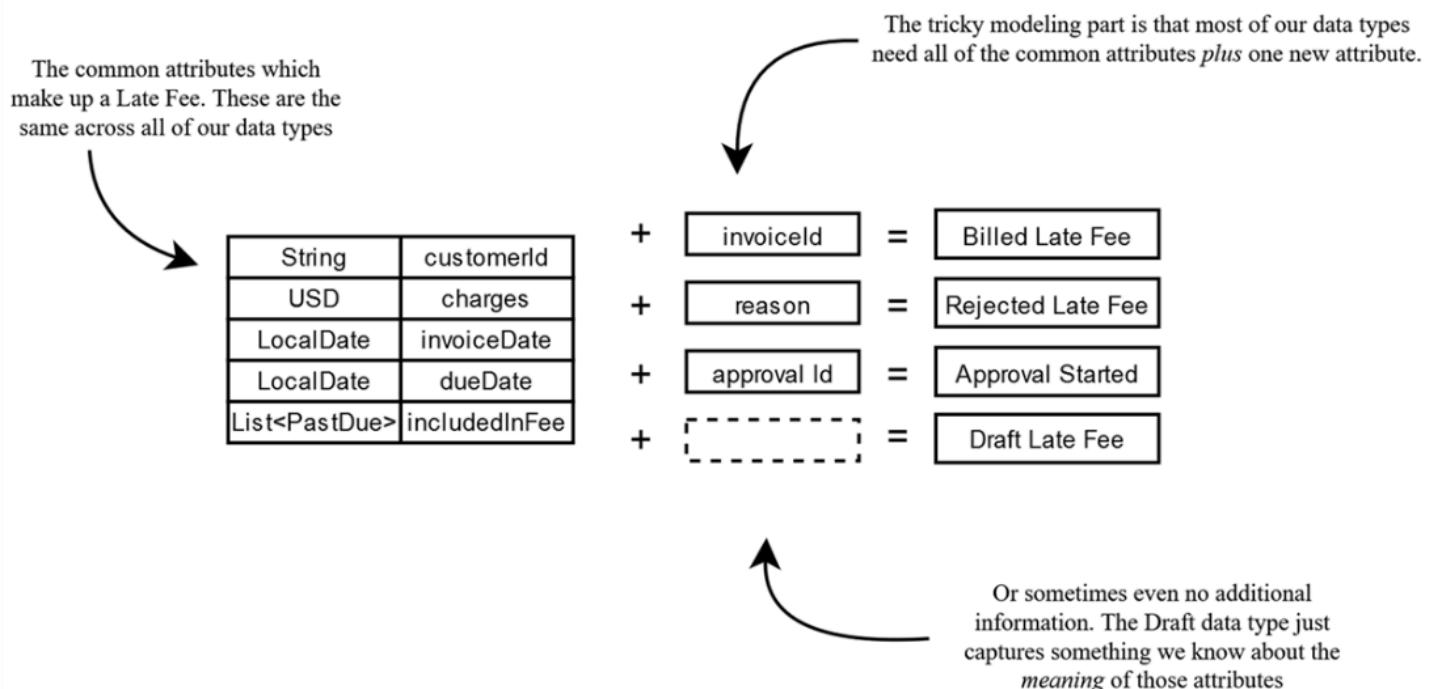


Figure 5.11 the challenge of modeling closely related data types without duplication

Records are great, but, by design, they don't allow extension. We can't inherit between data types, or derive new data types from existing ones. So, we have exactly two options: (a) duplicate the fields we need or (b) create wrappers.

The first thing we might try is to start with the simplest and "obvious" approach. We could design each type such that all of the attributes are duplicated between them.

Listing 5.26 One approach: duplicating all of the common fields across each data type

```
record DraftLateFee(  
    String customerId,           #A  
    USD total,                 #A  
    LocalDate invoiceDate,     #A  
    LocalDate dueDate,         #A  
    List<PastDue> includedInFee #A  
){}  
  
record BilledLatefee(  
    InvoiceId id,              #B  
    String customerId,  
    USD total,  
    LocalDate invoiceDate,  
    LocalDate dueDate,  
    List<PastDue> includedInFee  
){}  
  
record RejectedLatefee(  
    Reason reason,             #C  
    String customerId,  
    USD total,  
    LocalDate invoiceDate,  
    LocalDate dueDate,  
    List<PastDue> includedInFee  
){}  
  
record InReviewLatefee(  
    ApprovalId approvalId,      #C  
    String customerId,  
    USD total,  
    LocalDate invoiceDate,  
    LocalDate dueDate,  
    List<PastDue> includedInFee  
){}  

```

#A These attributes are common across every data type

#B Billed late fees only have one novel attribute: the invoice ID

#C Ditto for these. Other than this one extra piece of info, they're exactly the same as the others.

Most developers will, somewhat appropriately, have a visceral reaction to duplication like this. We're competing with the "free" usually supplied by our frameworks and ORMs. Slapping nullable fields onto an existing Entity is "easier" than maintaining this duplication. Comfort generally beats out semantics.

But before we dismiss this for not being DRY enough, let's look at what it enables. These data types are clunky in isolation, but expressive in the broader program. They make it possible to enforce semantic invariants.

Listing 5.27 Only Drafts can be assessed

```
ReviewedDraft assessDraft(DraftLateFee fee) {...} #A
```

#A Not an Invoice, or a Billed Late Fee, only a Draft late fee can be supplied to this method!

They're what enable us to tell a story with just data types.

Listing 5.28 Revisiting the narrative of our feature

```
List<Invoice>
-> List<PastDue>
-> DraftLateFee
-> Billable -> BilledLateFee OR RejectedLateFee
-> NeedsApproval -> InReviewLateFee
```

However, they remain riddled with duplication, and that won't do. We have to figure out how to get the semantic clarity without all of the copy and pasting.

So, as a next step, we might try extracting the duplicated code into its own data type so we can "reuse" it in all our models. Let's give that a shot and see how it feels.

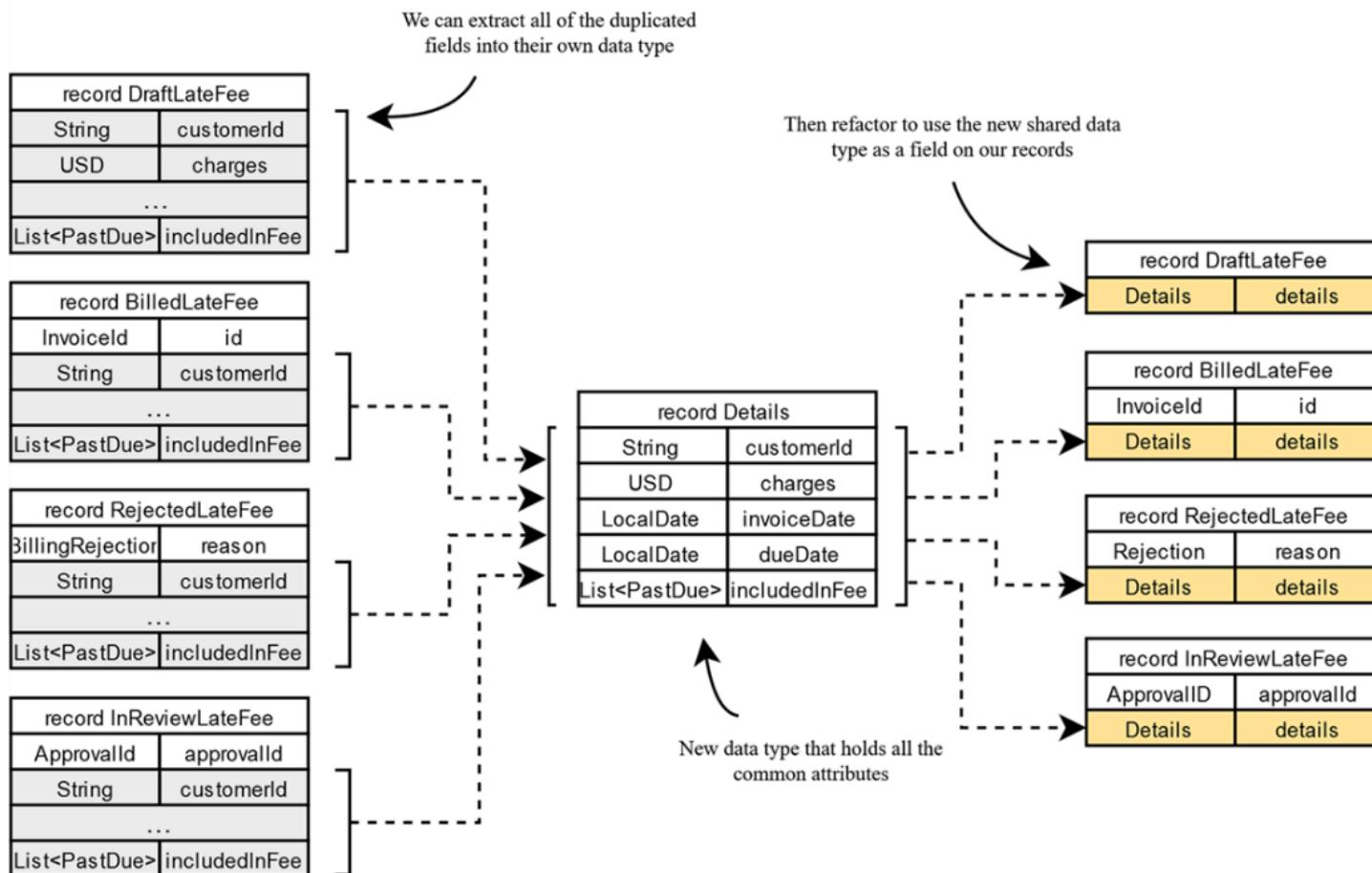


Figure 5.12 refactoring approach: extracting common fields into their own type

This looks pretty clean. However, there's a subtle downside lurking in this approach. You generally won't notice it until you start writing code that uses these types. They're extremely awkward to program with for anything outside of the narrow use case we've designed. Draft, Billed, and Rejected are all the exact same Late Fee, just in different states, but our modeling makes it very hard to treat them as one logical thing.

For instance, if we want to compute some aggregate statistics on all the late fees in our system, there's not really an easy way of doing that, because each of these types are isolated in their own little world.

Listing 5.29 I just want to compute some stats for my system! Why is this so hard?!

```
Map<???, USD> totalsByLifecycle(List<???> fees) {
    // ???
}
```

You might think that sealing them all together would alleviate these woes, but it actually doesn't. Java is blind to what kind of object it's dealing with until you either cast or pattern match it.

Listing 5.30 Sealing doesn't fix the awkward programming problems

```
sealed interface LateFee {                                #A
    record Draft (Details details) implements LateFee {}      #B
    record Billed (Details details, ...) implements LateFee {}
    record Rejected(Details details, ...) implements LateFee {}
    record InReview(Details details, ...) implements LateFee {}
}

Map<LateFee, USD> totalsByLifecycle(List<LateFee> fees) {
    fees.stream()
        .map(fee -> fee.details().total()) // NOPE #C
        // ...
}
```

#A Sealing our types from Figure 5.12

#B We're shortening the names since the "LateFee" part is now explicit in the interface

#C You might expect this to work because they all have the same data type, but to Java, they're just the interface "LateFee"

To Java, those sealed types are all just some random interface called "LateFee." It has no clue that there are records inside of it with attributes that we want to access. So, to make this very, very simple calculation work, you'd have to do some ad-hoc pattern matching in the middle of the computation just to get access to the attributes that you know all of them have.

Listing 5.31 A little ad-hoc pattern matching

```
Map<LateFee, USD> totalsByLifecycle(List<LateFee> fees) {
    fees.stream()
        .map(fee -> switch(fee) {
            case Draft d -> d.details();          #A
            case Billed b -> b.details();          #A
            case Rejected r -> r.details();        #A
            case InReview u -> u.details();        #A
        })
        // ...
}
```

#A Ugh...

You'll stumble into lots of these modeling woes as you take your first few steps with data-oriented programming. Often, data types can look beautifully descriptive at a high level, but fall apart once you start trying to use them in code. This is totally normal. You're trying something new. You'll get better at modeling over time.

What you generally don't want to do when you start feeling this friction is try push through those modeling problems by bolting on "fixes." For instance, faced with the

problem of Java being blind to what's inside of the sealed hierarchy, we could "fix" it by adding an interface that *tells* Java what we know about those types.

Listing 5.32 patching around our own modeling limitations

```
interface HasDetails { #A
    Details details();
}
sealed interface LateFee extends HasDetails { #B
    record Draft (Details details) implements LateFee {}
    record Billed (Details details, ...) implements LateFee {}
    record Rejected(Details details, ...) implements LateFee {}
    record InReview(Details details, ...) implements LateFee {}
}

Map<LateFee, USD> totalsByLifecycle(List<LateFee> fees) {
    fees.stream()
        .map(fee -> fee.details().total()) #C
        // ...
}
```

#A Ham-fisting in an interface that mirrors a record's accessor

#B Then having our LateFee interface extend it

#C Now we can access the details attribute without pattern matching. Hooray...?

But you (generally) shouldn't do that. Friction is valuable feedback. It's telling us that our current approach might not be the best.

These data types are the same Late Fee, just in different lifecycle phases. So, what we might try next is going in the opposite direction. Rather than factoring out what's *common*, we can factor out what's *unique*.

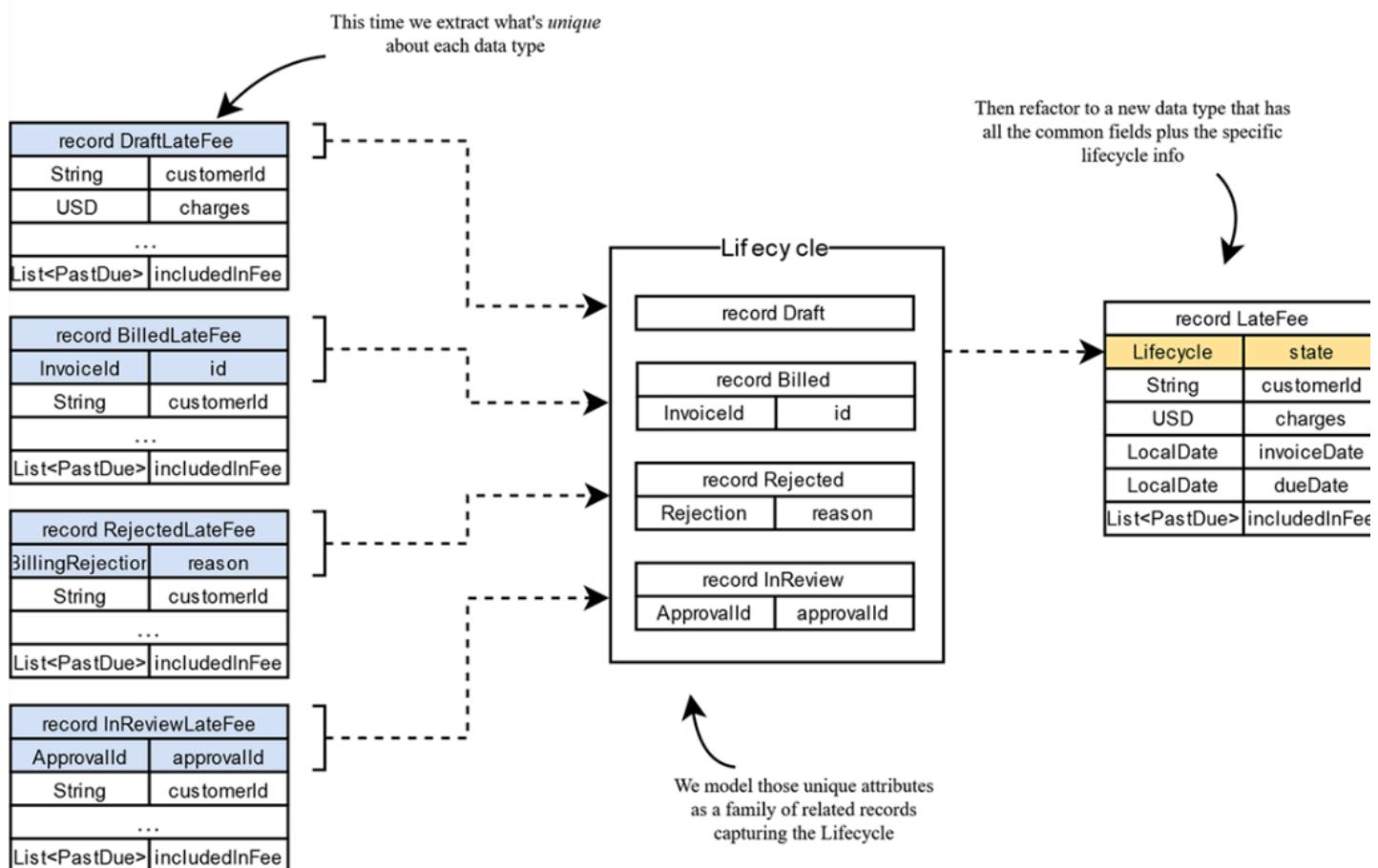


Figure 5.13 Another design option: refactoring to extract what's unique

This change gives an immediate improvement to the ergonomics of our data types. Their inherent “sameness” comes through, which makes them pleasant to program with again.

Listing 5.33 Now computing aggregate stats across related data types is easy!

```
Map<Lifecycle, USD> totalsByLifecycle(List<LateFee> fees) {
    return fees.stream()
        .collect(Collectors.groupingBy(LateFee::state,
            mapping(LateFee::total, reducing(USD.zero(), USD::add))));
```

However, this ergonomic improvement came at the expense of semantic clarity. This tug-of-war between semantics and ergonomics is a normal part of the design process. The lifecycle is now hidden away as runtime information, which means we're back on the hook for defending against the wrong states inside of our methods

Listing 5.34 Back to defensive programming to prevent incorrect billing

```
LateFee submitBill(LateFee fee) { #A
    if (!(fee.state() instanceof Lifecycle.Draft)) { #B
        throw new IllegalArgumentException("...");
    }
    // ...
}
```

#A We can no longer say what lifecycle is allowed, so we create lies in the code
#B It falls to defensive programming inside of the method

And because this information is hidden away, the “story” of our feature similarly becomes truncated and imprecise. We lose the ability to express how a late fee changes as it moves through our program.

Listing 5.35 Losing expressivity in our data flow

```
List<Invoice>
-> List<PastDue> #A
-> DraftLateFee##A
-> LateFee      #A
-> BillableFee -> LateFee (Billed OR Rejected) #B
```

#A We lose the subtlety of how the lifecycle of a late fee changes as it moves through our program
#B The code can only say that it works with “Late Fees”. Everything else is hidden at runtime.

Interesting, right? All we did was tweak the representation of our data, and it completely changed what we could express in the program as a whole.

The original factoring enabled us express interesting invariants in our system at compile time. However, this representation was clunky and “over-fit”. The data types were awkward to use outside of their narrowly designed use cases. Sometimes that’s what we want – it’s the very core of what “type safety” means: limiting what’s allowed. However, sometimes, like this time, it gets in the way.

So, the second refactoring moved that same information “down” a level. The lifecycle information became “just” an attribute on some *other* data type. This solved the over-fitting, and restored our ability to do useful stuff with the data types, but at the expense of type safety. The type no longer carried enough compile time information to participate in invariants.

This pull between flexibility and type safety is one of the main things we manage while designing. One is not better than the other. There’s no right or wrong. The important part is getting the constraints we want *in the places we want them*. Sometimes that will mean through type safety, sometimes that will mean through runtime defenses.

Is there a way we can get a little bit of each world? Compile time safety where we need it, paired with representations that still allow us lots of runtime flexibility?

Let’s try this: what if we refactored to turn that lifecycle information into a *type variable*?

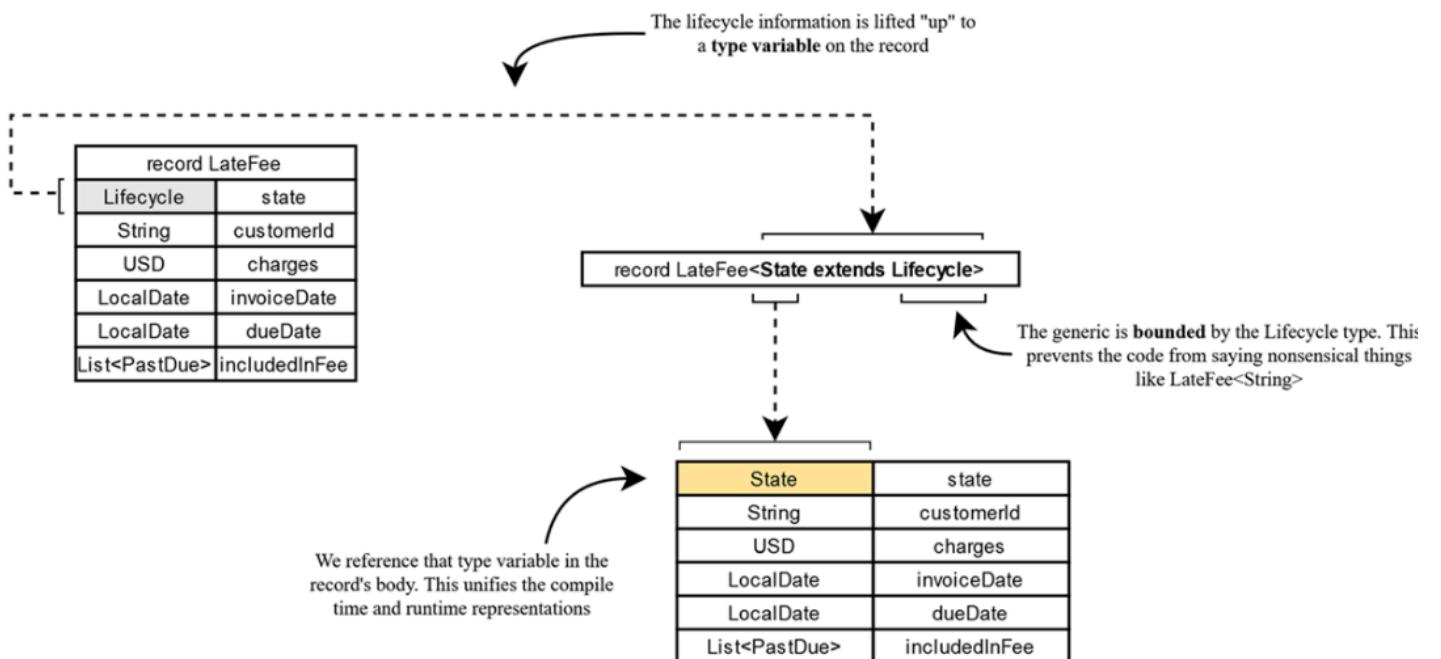


Figure 5.14 another approach: refactor to express semantic information as a type variable

The ability to *parameterize* data types is a powerful and underutilized capability in Java. It's very easy to program in the language for years without ever exploring how generics actually work or learning what they can express. Too often, their usage begins and ends with the collection types (`List<Integer>`), or built in interfaces (`Comparable<Integer>`), but generics offer a rich world of modeling opportunities. It gives us another tool for controlling "where" we express invariants in our domain. Parameterization allows us to take arbitrary runtime information and "lift it up" to compile time invariants.

Listing 5.36 Capturing lifecycle state as a generic

```

sealed interface Lifecycle {                                #A
    record Draft() implements Lifecycle {};
    record Billed(InvoiceId id) implements Lifecycle {};
    record Rejected(Reason reason) implements Lifecycle {};
    record InReview(ApprovalId approvalId) implements Lifecycle{};
}

record LateFee<State extends Lifecycle> (      #B
    State state,                                     #C
    CustomerId customerId,
    USD total,
    LocalDate invoiceDate,
    LocalDate dueDate,
    List<Invoice> includedInFee
) {}

```

#A This stays the same

#B But we modify the LateFee record to track that information as a generic. We've lifted this information "up" to the type level

#C This lets us say what's inside of this object at compile time

This moves the semantic information we care about into a *type variable* that we can control at compile time. We'd describe this refactoring as making `LateFee` *parameterized*

by its lifecycle state. What's powerful about this approach is that we can use that type information to enforce the exact lifecycle state our methods need.

Listing 5.37 behaviors can communicate and enforce their own invariants

```
void doSomething(LateFee<Draft> fee) {...}          #A
void doSomething(LateFee<Billed> fee) {...}          #A
void doSomething(LateFee<Rejected> fee) {...}         #A
void doSomething(LateFee<InReview> fee) {...}          #A
void doSomething(LateFee<? extends Lifecycle> fee) {...} #B
void doSomething(LateFee fee) {...}                     #C
```

#A Methods can demand data in whatever life cycle state they need

#B Or leave it completely unstated if they don't care

#C You can even leave off the generic entirely, which will produce a useful warning in Java. (This can be a great technique for slowly adding pockets of type safety to legacy code without needing buy in on a full refactor)

And because that lifecycle information is up at the type level, we're able to tell a rich story of how data changes as it moves through our application.

Listing 5.38 Restoring expressivity in our data flow

```
List<Invoice>
-> List<PastDue>
-> LateFee<Draft>                                #A
-> BillableFee -> LateFee<Billed> OR LateFee<Rejected> #A
-> NeedsApproval -> LateFee<InReview>            #A
```

#A We can watch LateFee's lifecycle information flow through the story

Of course, there's no Silver Bullet. This approach isn't without its share of drawbacks. Some will find type signatures like `LateFee<? extends Lifecycle>` excessively noisy and unergonomic (maybe even "complex"). Perfect rarely exists. Especially on a team.

Are there other options? Of course! There's no shortage of alternative approaches worthy of exploration. Type safety is a knob we can dial up and down as needed. However, this representation is the one we'll stick with in the book. It strikes a decent balance between ergonomics, semantics, and safety.

This technique of shifting defenses out of runtime checks and into compile time ones creates an additional (and very strong) layer of safety in our applications. Equally important is what it communicates. It serves as a warning sign to all that read the code. It makes it known that there's something special going on here. Something *important*. We're protecting this functionality from ourselves. Charging people money is a big deal. Leaning on the compiler for correctness shows everyone we're taking that responsibility seriously.

5.4.4 Modeling the decisions

Now that we have all of the lifecycle information available, modeling the remaining decisions in our program is pretty straight forward. These decisions are just more of the same. Some data types will just be a "name" and represent things we know, others carry some additional information by way of attributes.

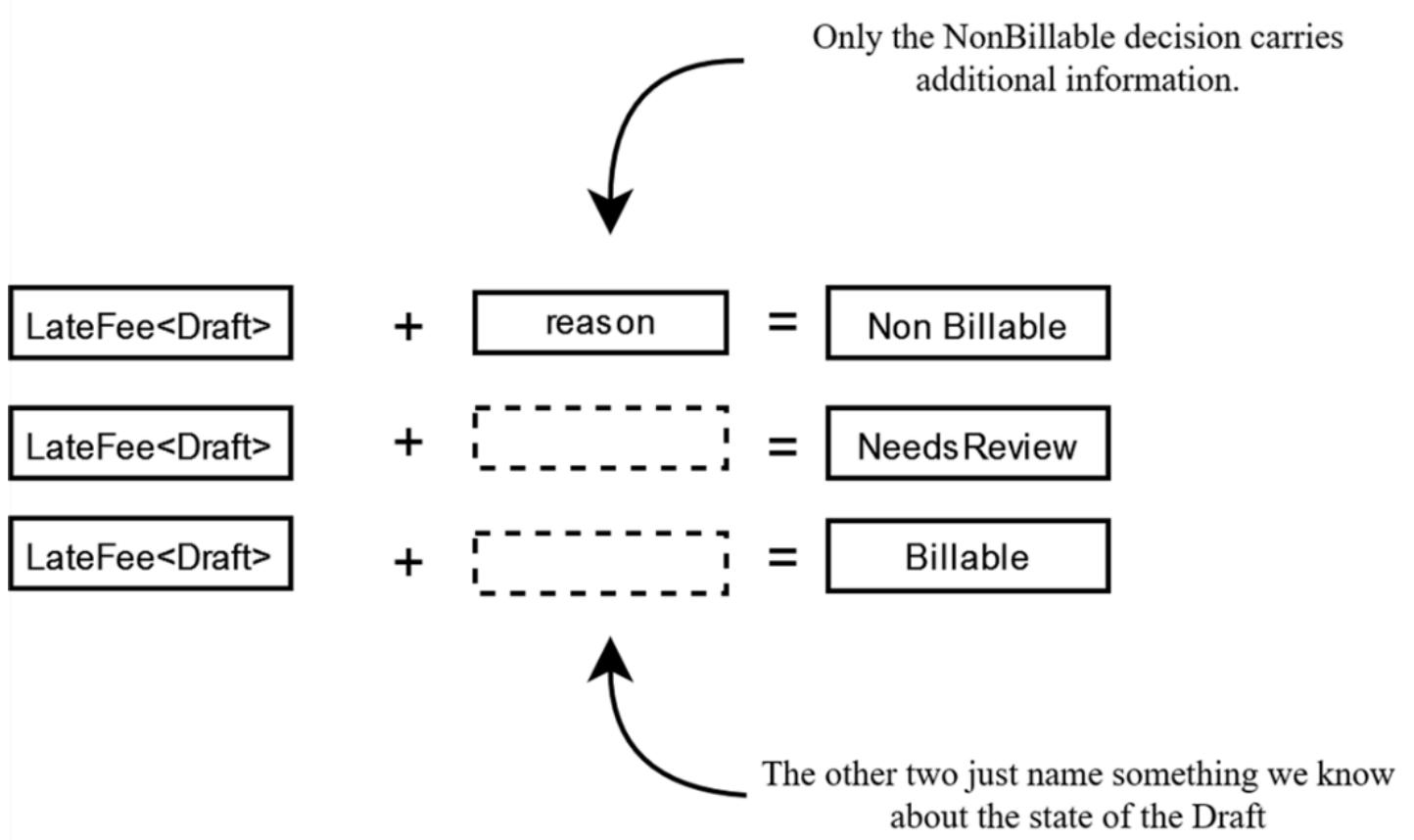


Figure 5.15 The different decisions for what we do with a draft

These decisions are modeling things we know about *draft* late fees. Not Billed or Rejected ones. So, this is where we can see our parameterization of `LateFee` begin to enable more cool things. This type level information doesn't just allow *methods* to say what lifecycle they care about, it allows other *data models* to do the same, too! We can encode the business rule that Assessments only deal with Drafts directly into the type system!

Listing 5.39 Representing the decision in our system

```
sealed interface ReviewedFee {
    record Billable(Latefee<Draft> fee) #A
        implements Decision{}
    record NeedsReview(Latefee<Draft> fee) #A
        implements Decision{}
    record NotBillable(Latefee<Draft> fee, Reason reason) #A
        implements Decision{}
}
```

#A Each of these decision data types specifies the exact life cycle that's relevant. No run time checks needed!

And that's it! Pretty easy, right? Like I said before, this process is several lengthy pages in a book, but represents only a few minutes of design effort in real life. Staying above the implementation allows us to rapidly iterate and explore how different designs and representations will affect our system before we commit to any code. That's pretty powerful.

5.4.5 The high-level data model

Let's bring it all together. Here's what we've got so far:

Listing 5.40 the current data model for our feature

```
public record PastDueInvoice(Invoice value) {}

public sealed interface Lifecycle {
    record Pending() implements Lifecycle {}
    record Billed(String invoiceId) implements Lifecycle {}
    record Rejected(Reason reason) implements Lifecycle {}
    record InReview(ApprovalId approvalId) implements Lifecycle {}
}

public record LateFee<State extends Lifecycle>(
    String customerId,
    USD total,
    State state,
    LocalDate invoiceDate,
    LocalDate dueDate,
    List<Invoice> includedInFee
) {}

sealed interface ReviewedFee {
    record Billable(LateFee<Draft> latefee)
        implements ReviewedFee {}
    record NeedsReview(LateFee<Draft> latefee)
        implements ReviewedFee {}
    record NotBillable(LateFee<Draft> latefee, Reason reason)
        implements ReviewedFee {}
}

List<PastDue> collectPastDue(List<Invoice> invoices) {...}
LateFee<Draft> buildDraft(List<PastDueInvoice> invoices) {...}
Decision assesDraft(LateFee<Draft> invoice) {...}
LateFee<? extends Lifecycle> submitBill(BillableFee draft){...}
LateFee<InReview> startApproval(NeedsReview needsReview) {...}
```

As you stare at Listing 5.40, I suspect the question of “is all of this worth it?” might be on your mind. There can be an initial sticker shock (your teammates will feel the same when you start pushing code). If we’re measuring against the “free” we usually get with frameworks and ORMs, our modeling has seemingly gone off the rails. We’ve “wasted” dozens of lines of code defining data types that could have been shared attributes on one of our existing entities (or maybe just a few new data types with many optional fields).

Designing around data can receive push back due to worries of “class explosions,” or “taxomania,” or even the appeals to original implementation’s “simplicity.”

We have to make sure we’re measuring the right thing.

Lines of code are not the enemy. Good data-oriented code is understandable. We spent those lines of code to buy something can fit in our heads. Those lines completely describe our domain.

They also defend it.

These data types have made a very dangerous part of our code safe by construction. The only way to send a bill in our service is to go through the effort of constructing a `LateFee`, provide it with the right `Lifecycle` state, and wrap all of that in the appropriate review type. Anything else and the code will not compile. This doesn't guarantee the absence of bugs (the type system can always be bypassed), but it greatly stacks the odds in our favor – all of this was purchased with a few dozen lines of custom data types.

Still, the question probably remains: do you have to do all of it? Would other approaches with fewer types be "lesser" somehow? Absolutely not! We went down this path because it let us explore some really interesting ways in which representation can enforce interesting invariants at compile time. It's how I would do it, but I index heavily towards type safety. You can dial the amount of typing up or down depending on need and (team) preference. Once you know that the power is there, you can decide where to apply it.

5.5 Fitting into the rest of the world

So far, we've designed our data types in a loose isolation of the rest of the world. This breathing room was nice while we were iterating on the design, but now we've got to figure out where all of those API calls, and database reads, and other external concerts fit into the world we've designed.

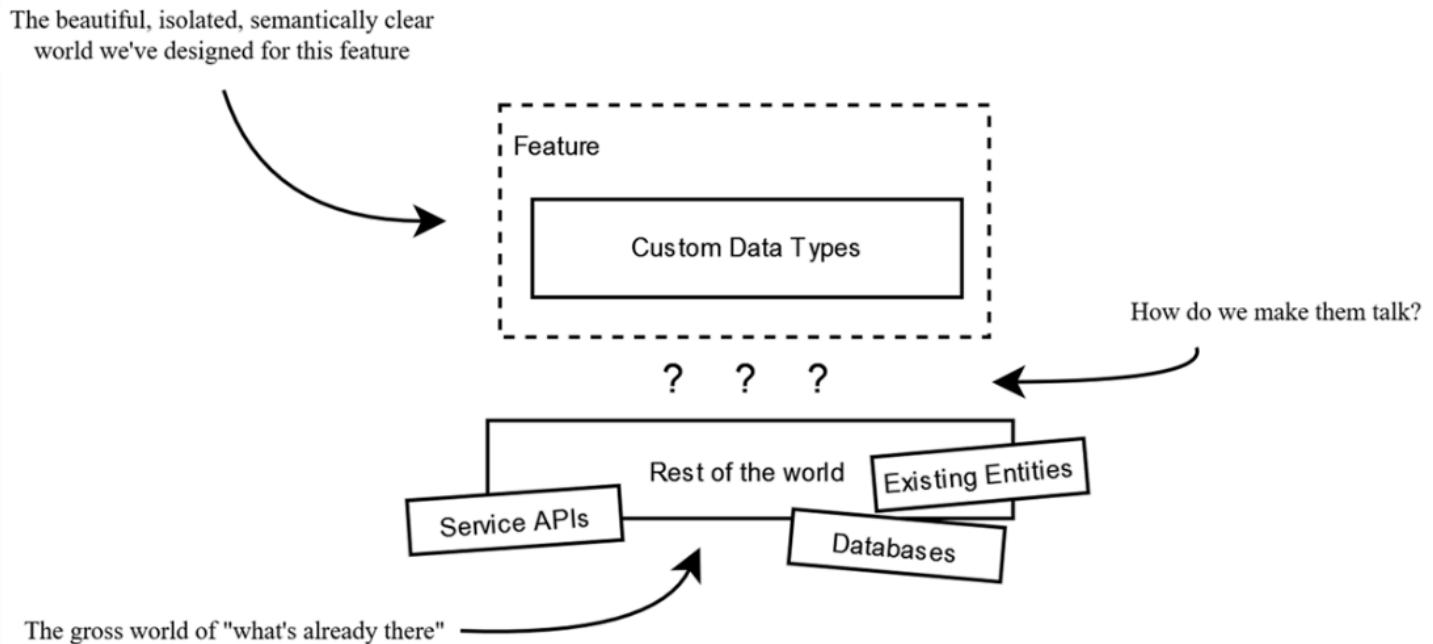


Figure 5.16 We have to figure out how to make the two worlds talk to each other

There's a sliding scale of options that goes something like this:

1. Fit our code to "what's already there"
2. Isolate the "how" from the "what"
3. Reshape the world as we want it to be

These are not mutually exclusive options. You'll regularly use all three (often mixing and matching within the same feature). One is not necessarily better than the other. You become more "data-oriented" as you go up the stack, because you gain deeper control over the semantics, but the tax you pay for that improvement is the need to make deeper cuts into the existing world. These are great refactorings, but expensive ones. In our day-to-day programming, we have to strike a balance between getting things done and crafting an idealized world.

5.5.1 Fitting into what's already there

The first approach we might try is to take our design and then sort of "slot" it into the rest of the world by extending each method with whatever we need.

Listing 5.41 Analyzing the dependencies of our methods

```
collectPastDue
  DependsOn:
    List<Invoice> (Entity / Database (read))
    CurrentDate (Environment (read))
    CustomerRating (API (read))
  Output:
    List<PastDueInvoice>

buildDraft
  DependsOn:
    List<PastDue> (Output from collectPastDue)
    CurrentDate (Environment (read))
    PaymentTerms (API (read))
    FeePercentage (Database (read))
    Customer (Database (read))
  Output:
    LateFee<Draft>

assessDraft
  DependsOn:
    LateFee<Draft> (Output from buildDraft)
    Rules (Database (read))
    Customer (Database (read))
    ApprovalStatus (API (read))
  Output:
    Billable
    OR NeedsReview
    OR NotBillable

submitBill
  DependsOn:
    BillableFee (output from AssessDraft)
    BillingService (API write)
  Output:
    BilledLateFee
    OR RejectedLateFee

startApproval
  DependsOn:
    NeedsReview (output from AssessDraft)
    CustomerID (database (read))
    ApprovalsService (API read/write)
```

The problem here is that each method depends on at least one service call, existing entity, or some kind of read from the external environment. After plugging in everything we needed, this design that used to be “ours” starts looking a little less familiar. It has grown, and most of what’s there isn’t ours anymore.

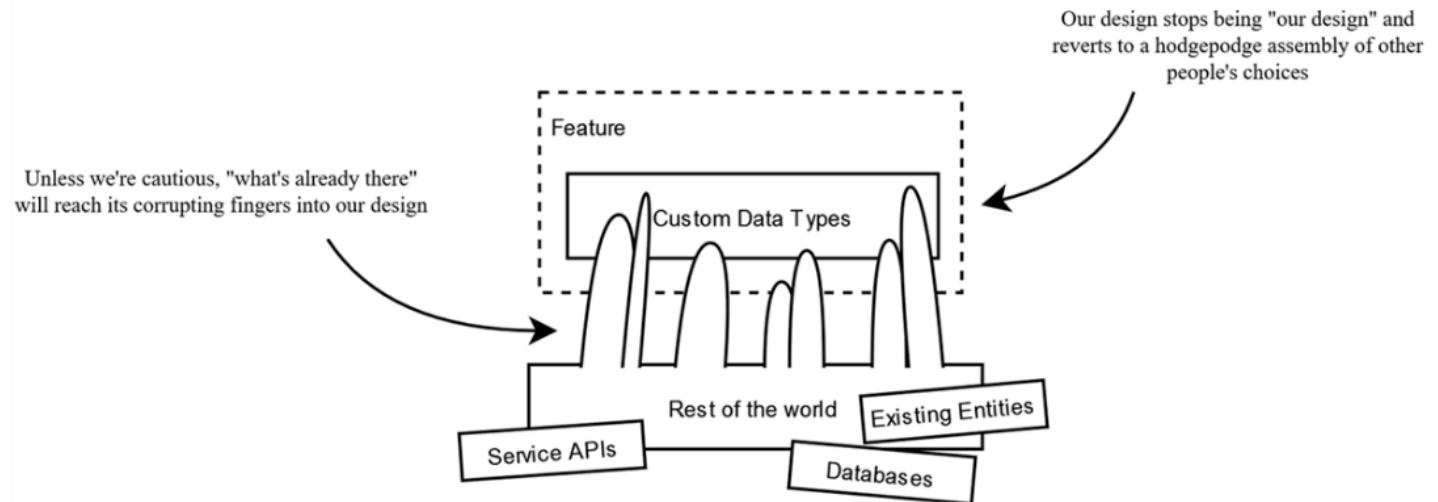


Figure 5.17 “what’s already there” shoving its sticky fingers into our world

This isn’t inherently bad, but it does warrant caution. It’s easy to be the frog slowly getting boiled as we allow more of other people’s design decisions to infect our own. Before we know it, we can be back in a world where our code becomes mostly about managing choices we never made. For instance, what might the implementation of `buildDraft` look like now that it’s riddled with external dependencies and concerns?

Something like this wouldn’t be unreasonable:

Listing 5.42 a possible implementation for buildDraft

```
Latefee<Draft> buildDraft(
    Customer customer,
    List<PastDue> invoices
){
    LocalDate today = LocalDate.now();          #A
    PaymentTerms terms = contractsApi.getTerms( #A
        customer.id()                         #A
    );
    BigDecimal feePercentage = feesRepo.get(   #A
        customer.billingAddress().country()    #A
    );
    // plus anything else we need...           #A

    #B
    return new Latefee<>(...);
}
```

#A The bulk of our methods will be devoted to managing the external world

#B Only once we've gotten through all of that can we start to write the business logic

But this isn’t great. The inside of our method ends up devoted almost entirely to the busy work of managing *how* we get the data. It claims to be a “draft builder,” yet it spends most of its time doing everything other than building a draft. It talks on the

network, reads from the database and environment, and who knows what else – the draft building ends up being the smallest part of what it does.

It might seem subtle or nit-picky, but this small lack of focus slowly compounds into code that cannot be trusted. It becomes something that forces you to skeptically read through every method to see what it *actually* does. The names and type signatures only act as loose starting points. These “little” things are what make code maintenance so difficult over the long haul. This is not the kind of code we want. We want to be able to read code and trust that it will do exactly what it says. No surprises.

The main problem with just plugging our design into the rest of the existing world is that we end up getting dragged all the way back to where we started: dealing with what’s already there.

We still need to merge these two worlds, so what we might try next is to join them together in a more measured way – one that decouples the details of how we get what we need from these existing services, from the details of what we do with it.

5.5.2 Separate how you get the data from what you do with it

Any method that calls an external dependency to read data can be refactored into one that *accepts* that data as an argument and offloads the fetching to some other method.

Listing 5.43 Refactoring to separate how we get data from how we use it

```
Latefee<Draft> doAction(){          #A
    // [LOAD DATA]           #A
    // [BUSINESS LOGIC]      #A
}

Latefee<Draft> doAction([DATA]){      #B
    // [BUSINESS LOGIC]      #B
}                                     #B
```

#A Any method like this, that handles both getting the data and doing something with it

#B Can be refactored into one like this. It accepts the data as input, rather than getting it itself.

It’s a small change, but a powerful one. If you do it for all of your methods, it ends up creating a nice delineation between “here’s where I get all my data” and “here’s what I do with it.” Those are very different problems, yet so frequently mixed together. We can use this dividing line to carve out exactly where we want our two worlds to interact.

Listing 5.44 a refactored buildDraft. No more “how” just “what”

```
Latefee<Draft> buildDraft(
    Customer customer,
    LocalDate today,
    PaymentTerms terms,
    BigDecimal feePercentage,
    List<PastDue> invoices
){
    // now I'm pure business logic!
}
```

This refactoring can be done for all of the methods which make up the “core” of our feature. Each pushes the problem of where the data comes from outside of themselves.

Each becomes a bit more isolated from the outside world in the process.

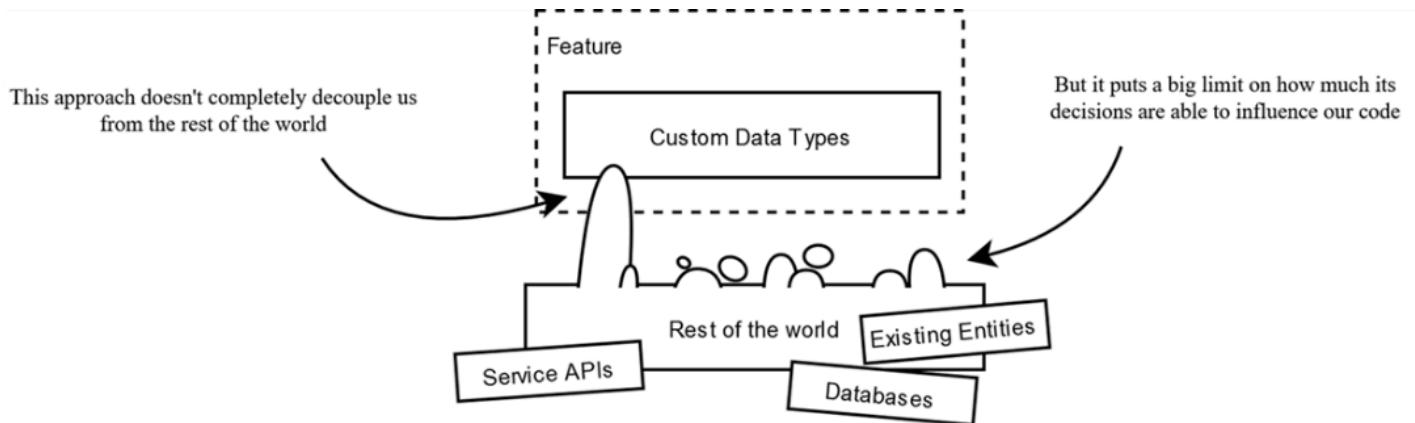


Figure 5.18 Limiting how many fingers can reach their way into our design

If you keep doing this refactoring, you'll eventually end up with two coarse buckets: one that gets the data, and one that does something with it. Neither needs to interact with the other.

Listing 5.45 Two buckets of different functionality

```
public void processLatefees() {  
    [LOAD ALL THE DATA WE NEED] #A  
    CountryCode country = customer.getBillingAddress().country();  
    BigDecimal feePercentage = feeRepo.get(country);  
    List<Invoice> allInvoices = invoiceRepo.findInvoices(customer.getId())  
    // and so on ...  
  
    [USE THE DATA IN BUSINESS LOGIC] #B  
    List<Invoice> pastDue = collectPastDue(allInvoices, rating, today)  
    // and so on...  
}
```

#A All of the data can be loaded in one spot in isolation

#B Then we can USE that data in our business logic. Neither side needs to know about the other!

This refactoring is far more powerful than it first appears. Decoupling the "how" from the "what" buys you atomicity. All of these network calls and database reads – side-effects which can fail for a near infinite number of arbitrary ways – all become quarantined to one spot in the code. They succeed together or they fail together. This makes reasoning easier. It makes testing easier (a topic we'll have a lot to say about later). It makes further refactoring easier.

Once you've split them apart in the same method, a pretty natural next question arises: why not go further and decouple them completely? "How we get the data" and the "what we do with it" are different responsibilities. We can manage each independently.

Listing 5.46 Isolating all the services calls and loading into its own method

```
record InvoicingData(          #A
    Customer customer,
    List<Invoice> invoices,
    LocalDate currentDate,
    CustomerRating customerRating,
    PaymentTerms terms,
    Percent feePercent,      #B
    Entities.Rules rules,
    Optional<Approval> approval
){}

class FeeService {
    ...
    public void processLateFees(){...}

    Stream<InvoicingData> loadInvoicingData() {  #C
        return customerRepo.findAll().stream()
            .map(customer -> {
                // Load everything here                      #D
                return new InvoicingData(
                    customer,
                    invoices,
                    ...
                );
            })
    }
}
```

#A Another data type to capture all of the stuff we need for invoice processing

#B This also gives us an opportunity to make individual attributes more semantic and self-describing

#C We can give ourselves a nice method for doing this data loading

#D We'd do all of our service calls, environment reads, and whatever IO we need right here – completely out of the way of the business logic

Now we can refactor our main method to use this new abstraction.

Listing 5.47 An example implementation using the new data grabber

```
public void generateLatefees() {
    for (InvoicingData data: this.loadInvoicingData()) {      #A
        List<Invoice> pastDue = collectPastDue(...);           #B
        LateFeeDraft draft = buildDraft(...);                   #B
        ReviewedDraft decision = assessDraft(...);             #B
        switch(decision) {
            case Billable -> ...
            case NeedsApproval -> ...
            case NotBillable -> ...
        }
    }
}
```

#A All of the complicated “how” involved in getting data is abstracted behind this single method

#B The rest of the method can be devoted to the logic of what it does with that data

This is a pretty huge improvement. The method is now almost pure business logic. It's easy to skim and see what it's doing now that it's not buried in data fetching and IO.

This is the pattern I've found most data-oriented features settle into. It's not perfect; we still have some quirks and warts incurred from the existing world, but it hits that "good enough" marker. Our code is safer and more expressive. You can see the narrative emerge. The data types tell a story of what our behaviors do.

However, there's still something unsatisfying about it. There's a "noise" in the implementation. Despite all of the decoupling, our code still lives in the shadow of the external world's design decisions. The data on which we operate is arbitrarily broken up by service boundaries. Those boundaries infect our methods. It forces them to be "bigger" than they need to be. They have to accept more arguments, from more sources, because there's no unifying, semantic view of what all this data *is*. To get to that, we have to draw some new boundaries in our applications.

5.6 Decoupling from the Tyranny of What's Already There

Software design is about creating the world as you want it to be, not as you found it. All of these services, and repositories, and entities are the debris of previous design decisions. They might be the way they are for good reasons, but those reasons aren't *our* reasons. Their representation actively harms what we're trying to do.

A pattern that I've found very useful for addressing this mismatch is to make the data access layers "smart." Or smarter than you may think is normal (or maybe even reasonable).

These "smarts" are a bit of a forgotten pattern. The ubiquity of frameworks and ORMs has turned what used to be a logical *tier* in the application into just another class in our projects that's auto-generated. They've been relegated to the role of single-purpose translation layers that handle the mapping details of whatever database technology is "hidden" on the other side. As a result, our service classes get stuck dealing with all of the disparate pieces.

The data-oriented view is a return to the classic purpose of a "data access layer": *abstraction*. It exists to vend us data in whatever shape we ask of it. What goes on behind the scenes to make that happen, whether it's one database call, thirty-seven service calls, or a hodgepodge of both (and more!) doesn't matter. That's why it's a whole *layer*. It insulates us from all of the gross realities and arbitrary boundaries of the modern software landscape.

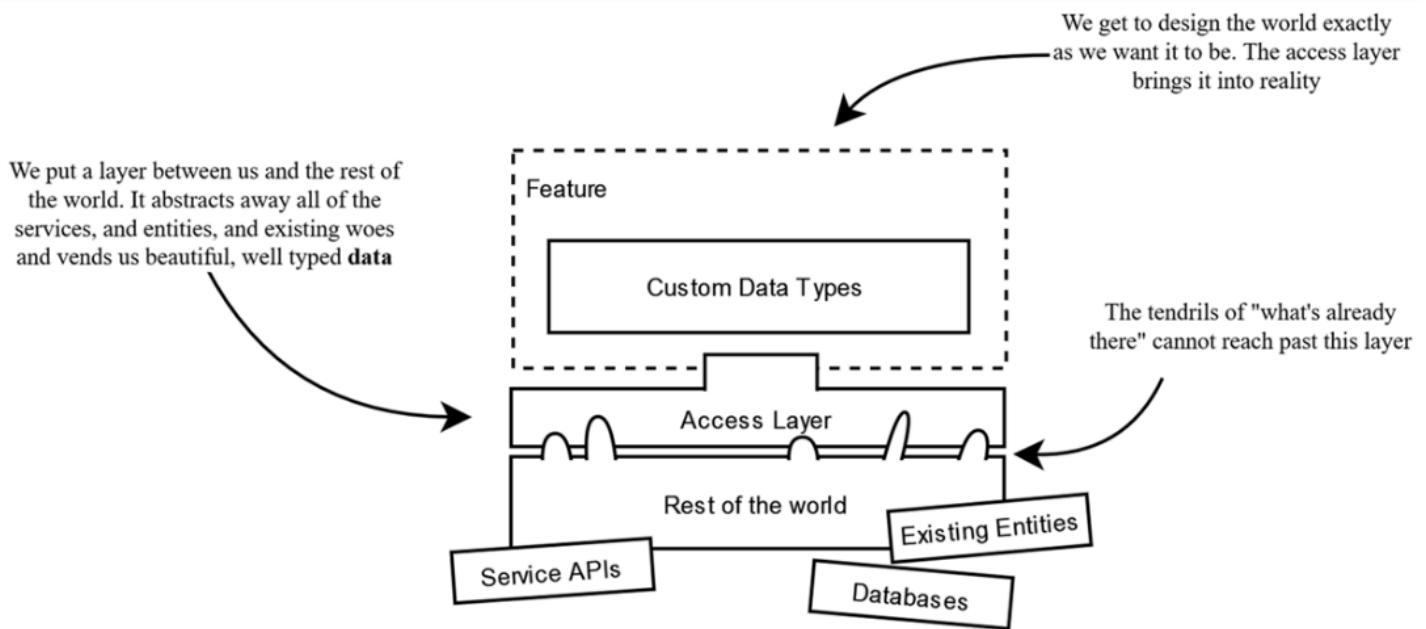


Figure 5.19 Good boundaries enable good design

This access layer is how we decouple from “what’s already there” and hide all of the things we don’t care about. The bulk of the service calls we make get data related to the customer. Payment Terms, Rating, etc. However, these end up as disconnected fragments of data floating around our application because nothing unifies them. The shape of the data mirrors the shape of the services which vend it.

To get rid of this problem, we just have to do some design. We can capture what all of these disparate service calls *mean* behind a new data type. We already have a Customer in the code base, but we can also create a new one for our needs, an *enriched* one that abstracts away all of the existing world.

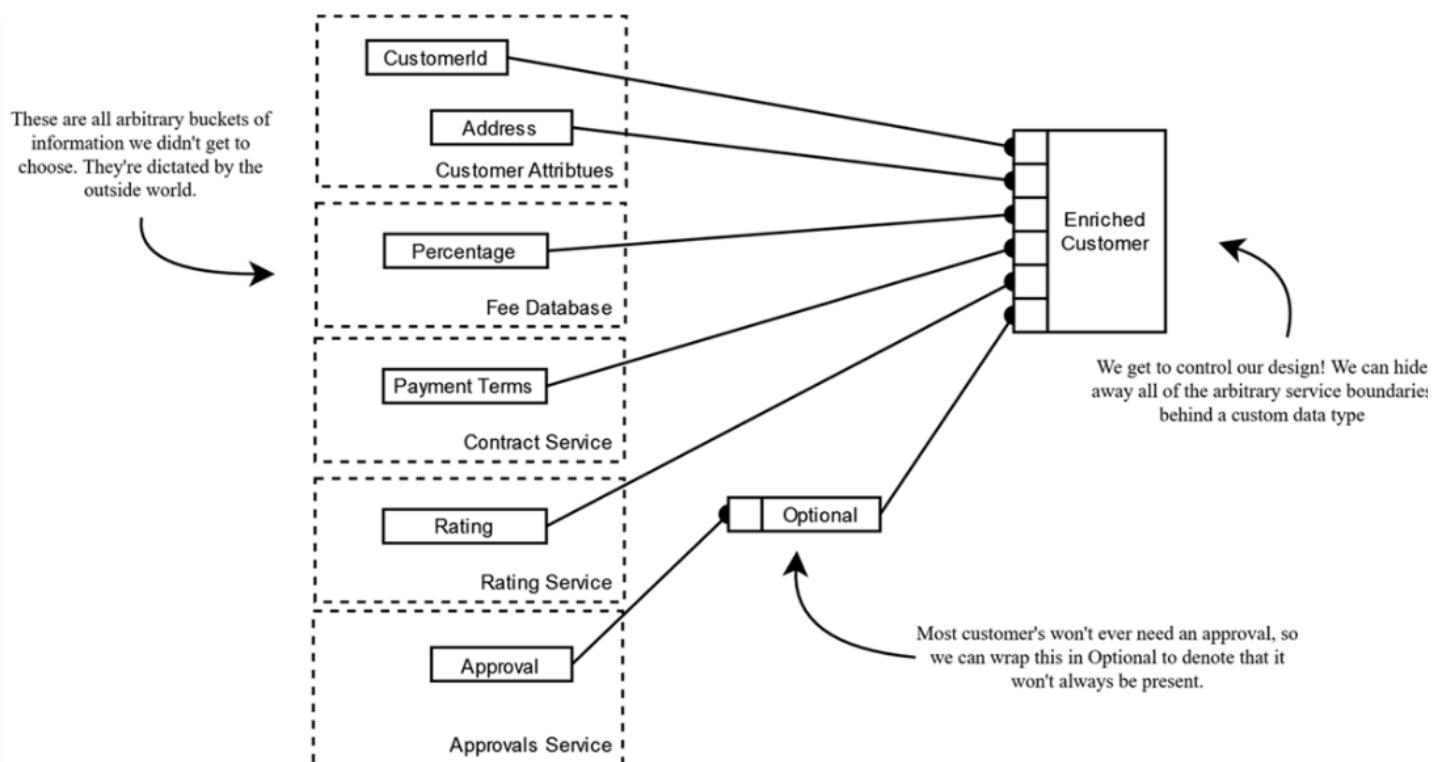


Figure 5.20 designing a new customer data type

This is the mindset of data-oriented programming. We're never done modeling. Always refining. Always trying to clarify what it is that this stuff in our programs means. The right data type can transform an entire program.

Listing 5.48 Enriched customer

```
record EnrichedCustomer(          #A
    CustomerId id,
    Address address,
    Percent feePercentage,
    PaymentTerms terms,
    CustomerRating rating,
    Optional<Approval> approval
){}
```

#A A new data type that hides all of the realities of the world from those who use it

Wrapped up in this simple record is a phenomenal amount of “abstraction” power. The construction of this data type will involve multiple service calls and database reads, but the consumers will never need to know any of that. This one data type unifies a sea of arbitrary service boundaries.

Now where do we put it?

This will be up to you. Many find it offensive if you have a “repository” (Data Access Object) do anything other than what the ORM blesses. Fights over “purity” are seldom worth having. A good middle ground tends to be introducing a façade to tie things together. We give ourselves a new entry point into retrieving these enriched customers. It does the grunt work of getting all this disparate information into a unified shape.

Listing 5.49 An improved access layer for retrieving customers

```
class StorageFacade { #A
    private CustomerRepo customerRepo; #B
    private ContractsAPI contractsApi; #B
    private RatingsApi ratingsAPI; #B
    private FeeRepo feeRepo #B
    private ApprovalsAPI approvalsAPI; #B

    public Stream<EnrichedCustomer> findAll() {
        return customerRepo.findAll().map(this::enrich);
    }

    private EnrichedCustomer enrich(Customer customer) { #C
        CountryCode country = customer.getBillingAddress().country();
        BigDecimal feePercentage = feeRepo.get(country);
        List<Invoice> allInvoices = invoiceRepo.findInvoices(customer.getId());
        CustomerRating rating = ratingsApi.get(customer.id());
        PaymentTerms terms = contractsApi.get(customer.id());
        String approvalId = customer.getApprovalId();
        Optional<ApprovalStatus> status = Objects.isNull(approvalId)
            ? Optional.empty()
            : Optional.of(this.approvalsApi.getApproval(approvalId))

        return new EnrichedCustomer(
            customer.id(),
            customer.address(),
            feePercentage,
            terms,
            rating,
            approval
        );
    }
}
```

#A This façade ties together a whole host of storage and service calls

#B All of these service boundaries are arbitrary and things our application doesn't care about. We want the data; we don't care about how.

#C This one method keeps so much noise out of the rest of our application.

This is, of course, just one of an infinite number of ways of slicing the problem.

Whatever encapsulation you ultimately choose, the end goal is that we can draw a line in our application that hides all of the stuff that the feature *doesn't need to care about*. This line should insulate us from the outside world.

5.7 The finished data model

Here's where we're at after all of our modeling.

Listing 5.50 The finished model

```
record PastDueInvoice(Invoice value) {}      #A

sealed interface Lifecycle {
    record Draft() implements Lifecycle {}
    record Billed(String invoiceId) implements Lifecycle {}
    record Rejected(Reason reason) implements Lifecycle {}
    record InReview(ApprovalId approvalId) implements Lifecycle {}
}

record Latefee<State extends Lifecycle>{      #A
    EnrichedCustomer customer,
    USD total,
    State state,
    LocalDate invoiceDate,
    LocalDate dueDate,
    List<PastDue> includedInFee
} {}

sealed interface ReveiwedFee {                  #A
    record BillableFee(Latefee<Draft> latefee)
        implements ReveiwedFee{}
    record NeedsReview(Latefee<Draft> latefee)
        implements ReveiwedFee{}
    record NotBillable(Latefee<Draft> latefee, Reason reason)
        implements ReveiwedFee{}
}
}

record EnrichedCustomer{                      #A
    String id,
    Address address,
    Percent feePercentage,
    PaymentTerms terms,
    CustomerRating rating,
    Optional<Approval> approval
} {}

// Behaviors
List<PastDue> collectPastDue(            #B
    EnrichedCustomer customer,
    List<Invoice> invoices
) {}

List<Latefee<Draft>> buildDraft(          #B
    EnrichedCustomer customer,
    List<PastDueInvoice> invoices
) {}

ReviewedFee assessDraft(                  #B
    Rules rules,
    EnrichedCustomer,
    Latefee<Draft>
) {}

Latefee<? extends Lifecycle> submitBill(   #B
    BillableFee draft
) {}

Latefee<InReview> startApproval(          #B
    NeedsApproval draft
) {}
```

#A The data for our feature

#B And all of the behaviors which operate on that data

This is our design. It's a world we created just for us. The ability to do this is one of the biggest powers of data-oriented programming. Plain data requires much less buy in than identity objects. Creating entirely new representations is cheap. If we don't like what's already in the code base, we can project it into a new shape – one that's purpose built for what we want to do. The only thing it costs is some mapping code at the boundaries.

But that mapping code isn't free, of course. It's still code we have to write. These new data types are more stuff to maintain. It might initially seem overwhelming -- especially compared to the "free" we usually get when slapping fields onto an existing Entity. It would be natural if you were still wondering: "is this all worth it?".

So, I want to give you two things.

The first is that much of what might make it seem like "a lot" is that you're seeing everything in the domain laid out in one place. We're not used to seeing it so clearly. The cost of legacy code is usually spread throughout the code base. We pay it in little increments here and there, little by little. Measured in aggregate, these "little" costs add up to great ones.

The second is this: if it seems like a lot, that's because it is! We're finally showing what's involved in this complex feature. The code is no longer able to lie or hide its requirements. It no longer offloads understandability to someone else. What we're seeing is what's always been hidden there. We've just brought it into the light.

5.8 Wrapping up

This is data-oriented programming in the real world. It's messy and filled with little concessions and hard choices, but focusing on data gives us the tools to punch through ambiguity and lift our requirements up to the top where everyone can see them. Small tweaks to the algebra of our data gives us immense control over where and how strongly we can enforce invariants.

Next up, we're going to implement this model. Java 21 has lots of exciting tooling for us to explore.

5.9 Summary

- Design is about creating the world as you want it to be, not as you found it.
- Reuse can be harmful. Services and Entities designed for other use cases will seldom fit ours.
- Assembling behaviors out of ill-fitting representations *creates* illegal states and potential bugs
- It's OK to introduce new data types into your system!
- Good data-oriented design uses data types to tell a story about the requirements
- The design process is "done" when the code looks like it's for humans rather than machines
- Design is iterative. The first option is just the first option. Keep exploring.

- The same data can be represented in an endless number of ways. Try multiple factorings
- Good representations strike a balance between semantics, ergonomics, and safety
- Behaviors are modeled as series of factory-like data transformations. Data in; data out.
- Ignore implementation details and work “above” the code while designing. This enables rapid iteration.
- Simple wrapper types are a good option for capturing “something we know” about a particular state
- Avoid the pursuit of purity. The real-world is messy. Our modeling is in pursuit of clarity
- We can interact with identity objects from a data-oriented context as long as we’re cautious. Wrapping in a record can serve as a light quarantine for mutability.
- The amount of type safety we employ is a design decision
- We have a responsibility to the people who use our software. The higher the stakes, the more we should weigh moving invariants into compile time checks.
- Generics aren’t just for collections!
- Type variables can be used to convert runtime invariants into compile time ones
- Good modelling can make illegal behaviors impossible to express
- If / else conditions at runtime can be expressed as algebraic sum types at compile time
- Decoupling decision from action lets us decide when (if at all) that action gets performed
- A lot of care is needed when fitting our design into the rest of the world
- Without caution, our design can be infected with “what’s already there”
- Separate how you get the data from what you do with it.
- Decoupling “how” from “what” brings atomicity to your data loading and simplifies failure modes
- Use a Data Access Layer to hide the details of existing systems and vend well-formed semantic data
- The Façade pattern is a good option for creating “Smart” Repositories / “Data Access Objects”

6 Implementing the Domain Model

This chapter covers

- Implementing the domain model
- Functions and methods
- Designing around determinism

It's time to start writing some code! This chapter picks up where we left off in Chapter 5. We've finished modeling a complex invoicing domain that charges late fees. All of the data and behaviors have been sketched out. Now it's up to us to implement them.

This requires care. A bad implementation can undermine a good data model. Methods in Java are up to no good most of the time. We're going to learn how to get them under control.

We're also going to spend time figuring out where to put all of this implementation code. So far, our modeling has been purposefully indifferent to details like class ownership. We focused only on the data and how it flowed through the program. But now we need to make architectural choices. Who should own these behaviors? Who can they talk to? Who can talk to them?

We're going to learn how a really simple idea can transform both how we implement our methods and how we organize them into classes.

6.1 On the criteria by which we break down an implementation

In the last chapter, we tweaked the design of our methods with a simple heuristic: separating how we get the data from what we do with it.

This was done for selfish reasons. We wanted to keep the tendrils of "what's already there" out of the clean new world we were designing. So, if a method reached out to some external service, we refactored to move that responsibility elsewhere. Each method was simplified until the only thing it was allowed to do was accept data as input and return data as output. Those ingress and egress points became their only connection to the world. They no longer knew about services, or what date it was, or where any of the data it receives comes from. Their world is inputs and outputs.

This refactoring caused an interesting property to emerge in our methods as we slowly cut them off from the outside world.

They became *deterministic*.

They became *functions*.

6.2 Methods, Functions, and Determinism

"Method" and "function" often get used interchangeably in Java to describe behaviors, but we're going to draw a semantic distinction between the two. This will take some unwinding, because Java makes no *syntactic* distinction between these two ideas.

Listing 6.1 Is this a method or a function?

```
class Example {  
    public Integer plusOne(Integer x) { #A  
        return x + 1; #A  
    } #A  
    Function<Integer, Integer> plusSomething = #B  
        (x) -> x + new Random().nextInt(); #B  
}
```

#A Is this a method? A function? Both?

#B What about this one?

So, we have to separate what we *say*, what we *mean*, and how we *represent* it in Java.

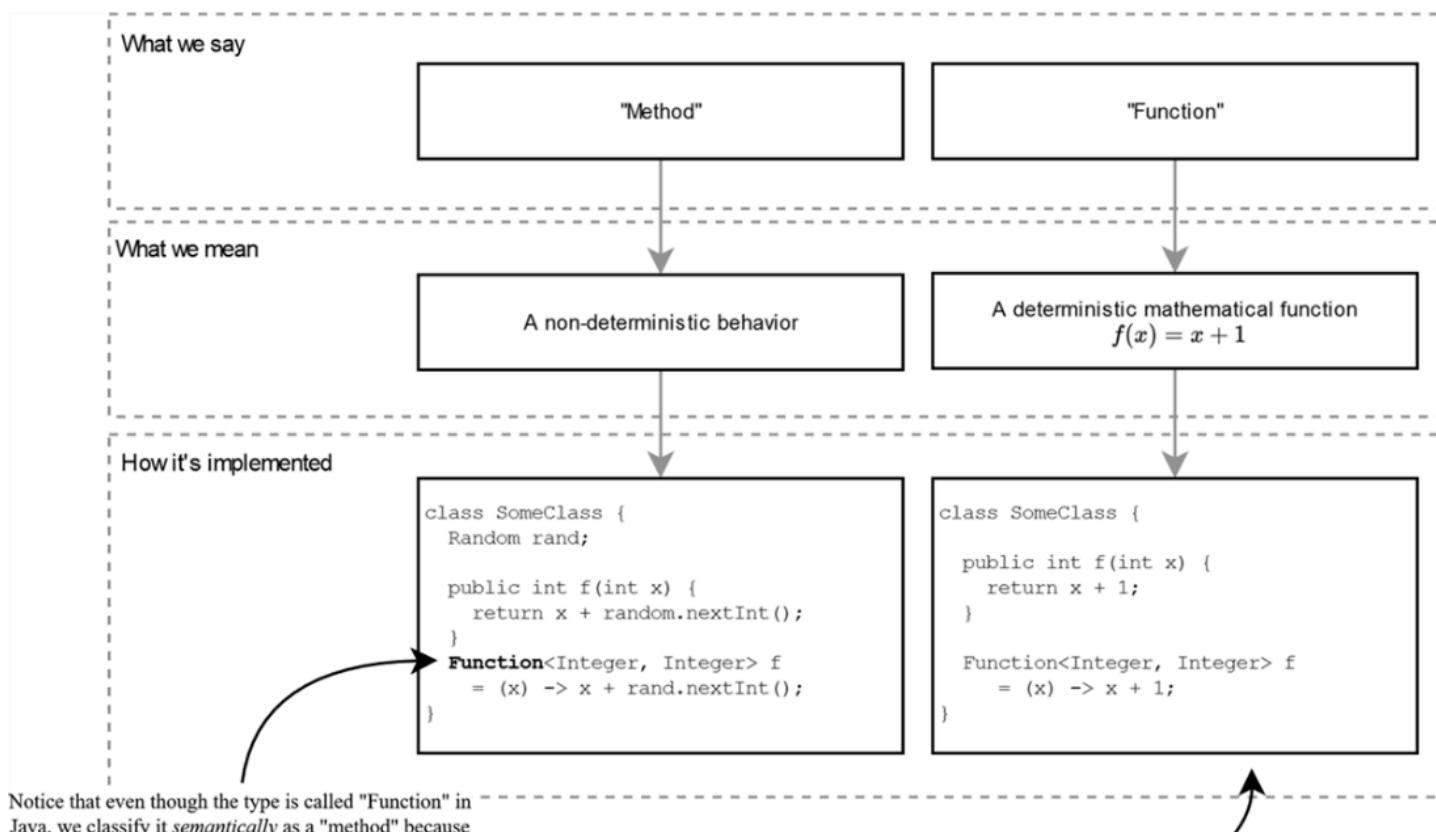


Figure 6.1 The difference between what we say, its semantics, and its implementation in Java

From now on, when we use the word "function," we're talking about it in the *semantic* sense and *not* talking about any particular Java syntax or type. Functions are **deterministic** mappings from inputs to outputs. It doesn't matter how we represent them. If they're deterministic, they meet our semantic meaning for the word "function."

This semantic difference matters because everything in Java is technically a “method” from the language’s syntactic lingo point of view. Methods are any behavior attached to a class, and *everything* in Java is a class behind the scenes, so every behavior is a method. This is true even for things we call “anonymous functions.”

Listing 6.2 Anonymous “functions” are just classes with methods behind the scenes

```
Stream.of(1,2,3).map((x) -> x + 1).toList() #A  
  
Stream.of(1,2,3).map(  
    new Function<Integer, Integer>() { #B  
        @Override #B  
        public Integer apply(Integer x) { #B  
            return x + 1; #B  
        } #B  
    } #B  
).toList()
```

#A In Java, we call this an “anonymous function”

#B But it de-sugars to an Anonymous Class with an “apply” method behind the scenes

So, when we’re talking about functions from now on, we’re talking about their meaning outside of the code. Their defining property is *determinism*. Inside of the code, we can represent functions with anything, because the *syntax doesn’t matter*. If it deterministically maps inputs to outputs, we’ll call it a function.

Listing 6.3 Considering the semantics independently of the syntax

```
class Example {  
    public Integer plusOne(Integer x) { #A  
        return x + 1; #A  
    } #A  
    Function<Integer, Integer> plusSomething = #B  
        (x) -> x + new Random().nextInt(); #B  
}
```

#A We call this one a function because it’s completely deterministic

#B This one, despite having the type “Function” in Java, would not be a semantic function because of its non-deterministic use of random numbers

In functional programming circles, this distinction gets referred to with words like “purity” and (if it meets a few other criteria) “referential transparency.” Functions with no external side-effects or dependencies are “pure”; functions (methods) with side-effects are “impure.” These are useful terms to know, but we’re going to shy away from them in this book. Instead, we’ll focus on the property we care about: *determinism*.

Determinism gives us another vantage point to explore how we design our software. It gives us a concrete criterion by which we can decompose our implementation. The “coupling” we used in the previous chapter is a good starting point, but it’s subjective. “Coupled” will mean different things to different people – plus, coupling isn’t inherently evil.

“Is it deterministic?” is an objective measure. It’s something we can ask about any method we write. If that method isn’t deterministic, a good follow up question is “can it be?” and after that: “should it be?” (the answer won’t always be “yes”).

Listing 6.4 Is this original implementation from Chapter 05 deterministic?

```
private LocalDate figureOutDueDate() { #A
    LocalDate today = LocalDate.now(); #B
    PaymentTerms terms = contractsClient.getTerms( #C
        invoice.getCustomerId()
    );
    switch (terms) { #D
        case PaymentTerms.NET_30:
            return today.plusDays(30);
        // etc...
    }
}
```

#A We can judge this implementation based on how its implementation choices affect its determinism
#B This choice makes it non-deterministic. The behavior will change based on when it runs
#C Same thing here. The decision our code makes is completely dependent on the whims of some other system
#D But once you get past all that, the method becomes completely deterministic. It maps that data to outputs.

This distinction matters, because non-determinism makes everything harder. If your implementation depends on another service or identity, it becomes up to the whims of those other parts of the system how *this one* will behave. Which means we, the poor people reasoning about the code, have to know about those whims, and, most obnoxiously, control for those whims in our tests.

Listing 6.5 Controlling for the whims of other parts of the system

```
@Test
void testFigureOutDueDate() { #A
    ContractsAPI mockApi = mock(ContractsAPI.class); #B
    when(mockAPI.getRating()).thenReturn(PaymentTerms.NET_30)

    LocalDate dateMock = mockStatic(LocalDate.class);
    when(dateMock.now()).thenReturn(LocalDate.of(2024, 1, 1));

    FeeProcessingService service = new FeeProcessingService( #C
        mockApi,
        // plus anything else we need...
    );

    List<Invoice> result = service.figureOutDueDate(); #D
    Assertions.assert(...);

    #E
}
```

#A Our test is doomed to be devoted to setting up mocks and stubs to control our dependencies
#B (or, you could imagine implementing the interface if that's your style)
#C In addition to the mocks, we also have to instantiate the entire object just to test part of it
#D We have to make it all the way down here before we call the thing under test
#E Plus, all that work just covers the happy path! Referencing dependencies means caring about when they fail.

This is what makes determinism a useful design tool. It can guide us towards implementations that are easier to reason about and test. It's the real reason our

simple heuristic from chapter 5 was so powerful. Separating how we got the data from what we did with it converted non-deterministic *methods* into deterministic *functions*.

Listing 6.6 Converting to a deterministic function

```
private LocalDate figureOutDueDate(
    LocalDate today,                                #A
    PaymentTerms terms                             #A
) {
    LocalDate today = LocalDate.now(); #A
    PaymentTerms terms = contractsClient.getTerms(...); #A

    switch (terms) {                                #B
        case PaymentTerms.NET_30:
            return today.plusDays(30);
            // etc...
    }
}
```

#A Refactoring to remove the dependencies which were robbing us of determinism. Now we take the data we need as input.

#B And everything else is deterministic. Same inputs. Same outputs. Every time. Forever.

It's a small change, but codebases are the sum of these choices. Good modeling ripples throughout the code. Testing deterministic code is dead simple.

Listing 6.7 Functions make tests simple

```
@Test
void testFigureOutDueDate() {
    #A
    List<Invoice> result = FeeService.figureOutDueDate(
        LocalDate.of(2024, 11, 17),   #B
        PaymentTerms.NET_30          #B
    );
    Assertions.assert(...);
}
```

#A No more mocks or test doubles.

#B We can test the logic by just passing around plain data

How much nicer is that? I'm a fan of anything that gets me out of writing mocks or test doubles. This is the power of functions.

6.2.1 How do we write functions in Java?

Java has no mechanism for enforcing that a method behaves like a deterministic function. So, it's mostly up to convention and discipline. The convention I like, and what we'll use in this book, is to make methods **static** whenever they're acting as deterministic functions. This is completely arbitrary, but it works pretty well as a signaling mechanism.

Listing 6.8 Using static to denote functions

```
public int increment(int x) {          #A
    return x + 1;
}

public static int increment(int x) {   #A
    return x + 1;
}
```

#A Both of these are pure deterministic functions. The static modifier helps communicate that we want them to stay that way.

Making methods static also acts as a natural hedge against the slow creep of ever-changing requirements. Methods which start out pure and deterministic have a way of becoming less so as different hands work on the project. Instance methods are only ever a few keystrokes away from non-deterministic interactions.

Listing 6.9 The constant allure. If it's in scope, someone will reach for it

```
class FeeService {
    private RatingsAPI ratingsApi;

    LocalDate figureOutDueDate(LocalDate today, PaymentTerms terms) { #A
        return switch(terms) {
            // [beautiful, clean, determinisitic implementation here]
        }
    }

    LocalDate figureOutDueDateV2(LocalDate today, PaymentTerms terms) {
        Rating rating = this.ratingApi.getRating() #B
        return switch(terms) {
            // use the result here.
            ...
        }
    }
}
```

#A This method is acting as a deterministic function, but there's nothing that cues to the people who work on our code that we want it to stay that way

#B so this becomes a very "obvious" change to make if we pick up a requirement that due Dates should also depend on the customer's current rating. Talking to the ratings API is an easy addition, but one that destroys the deterministic nature of the function.

Writing deterministic functions takes discipline. Maintaining them takes vigilance. Static modifiers exert a little pressure in the right direction.

Listing 6.10 Purposefully adding friction to the wrong path

```
class FeeService {  
    RatingsAPI ratingsAPI;  
  
    static LocalDate figureOutDueDate( #A  
        LocalDate today,  
        PaymentTerms terms  
    ){  
        // this.ratingsAPI Nope! #B  
        // ...  
    }  
}
```

#A Marking the method as static

#B The ratingsApi is now totally out of scope. We couldn't use it even if we wanted to. The function's determinism is protected. If we want that data, it'll have to be passed in explicitly

Of course, this isn't a perfect solution. Static things can just as easily use or reference things outside of them, but the loose convention that they *shouldn't* is easy to socialize across a team and enforce during code review.

6.2.2 The effect of deterministic functions on reasoning

Something unexpected happens when we put constraints on our methods. It seems backwards, but the less we allow methods to do, the more expressive they become. Functions are expressive *because* they're less powerful. They're only allowed to do one thing, and it's exactly what the types dictate.

To see this, let's start with something that's *not* a function and ask a slightly odd question: how many different implementations exist for this type signature?

Listing 6.11 How many ways could we write an implementation for this?

```
enum People {Bob, Mary}  
enum Jobs {Cook, Engineer}  
  
class SomeClass {  
    // [Hidden]  
  
    People someMethod(Jobs job) {  
        // [Hidden] #A  
    }  
}
```

#A What can we guess about what goes on inside of here?

The question itself might not make sense at first (what do you mean "how many"?). Ultimately, there can only be one implementation in the code, but the question is not about the one we *do* write, it's about all of the ones we *could have written*.

Inside of every method is an unconstrained, infinite space of *possible* implementations.

A method's name tells us what it *should* do. But names can, and frequently do, lie. Usually not at first, but slowly over time as their scope gets expanded by different hands. (How many times have we all seen a sudden spike in latency only to discover someone snuck a database call somewhere it didn't belong?)

Methods have no constraints. Their type signature and naming let us make a guess, but *only* a guess. The only way to know for sure is to read its full implementation.

Listing 6.12 One of an infinite set of possible implementations

```
enum People {Bob, Mary}
enum Jobs {Cook, Engineer}

class SomeClass {
    // [Hidden]

    People someMethod(Jobs job) {          #A
        return this.jobsRepo.findPerson(job) #A
    }
}

// Or maybe...
class SomeClass {
    // [Hidden]

    People someMethod(Jobs job) {
        this.setCombobulator("large");      #B
        universe.runSimulation(job);       #B
        collapseSingularity(this);         #B
        return getChosenOne();             #B
    }

    // Or it could...
    // Or maybe...
    // And so on off into infinity
}
```

#A Maybe it looks like this?

#B Or like this! Methods can engage in any kind of nonsense because they aren't constrained by anything. Their names and types are only a hint.

But let's make that same method a *function* using our static convention.

Now what can it do? How many possible implementations can we write for this?

Listing 6.13 using static to signal that this should be a deterministic function

```
enum People {Bob, Mary}
enum Jobs {Cook, Engineer}

static People someMethod(Jobs job) { #A
    // [Hidden]
}
```

#A We're using static to signal that this method will behave as a deterministic function

The question suddenly has a very different answer. Functions constrain what implementations are *possible*. There are only so many ways we can map an input to an output. Eventually we just run out of options.

Listing 6.14 Every possible implementation allowed by our types.

```
enum People {Bob, Mary};  
enum Jobs {Chef, Engineer};  
  
static People someMethod(Jobs job) {      #A  
    switch(job) {  
        case Chef -> People.Bob;  
        case Engineer -> People.Bob;  
    }  
// OR  
static People someMethod(Jobs job) {      #B  
    switch(job) {  
        case Chef -> People.Mary;  
        case Engineer -> People.Mary;  
    }  
// OR  
static People someMethod(Jobs job) {  
    switch(job) {                      #C  
        case Chef -> People.Bob;  
        case Engineer -> People.Mary;  
    }  
// OR  
static People someMethod(Jobs job) {  
    switch(job) {                      #D  
        case Chef -> People.Mary;  
        case Engineer -> People.Bob;  
    }  
// That's it!                         #E  
}
```

#A Implementation possibility #1

#B Implementation possibility #2

#C Implementation possibility #3

#D Implementation possibility #4

#E No other possible implementations! (we're ignoring cases where the input is ignored and only a fixed value is returned)

That's every possible mapping from inputs to outputs. You could implement the individual lines differently, or trade `switch` for `if` statements, add some indirection or whatever, but there are no other *behaviors* – no mapping from inputs to outputs -- that this function could express. The space of things it *could* do is completely bound by its types.

Thinking about things this way might seem strangely disconnected from reality right up until you sit down to write a test. Then you'll notice something. The surface area you need to cover is usually finite. In fact, it's often *small*. Exhaustible even. You can prove through simple, exhaustive brute force that your code correctly maps every possible input it to every possible output.

This realization is where the addicting part starts.

You can combine functions. The combination of two deterministic functions is a new bigger, more capable deterministic function. Small things can be combined into bigger things. And bigger after that. You can model entire features (and sometimes entire programs!) as combinations of deterministic, *provably correct* functions.

Listing 6.15 Combining deterministic functions produces new deterministic functions

```
var anotherDeterministicFunction =  
  collectPastDue.andThen(buildDraft); #A
```

#A if collectPastDue and buildDraft are both deterministic functions, then their combination is also a deterministic function!

But there are limits, of course. In most programs we'll need to interact with the outside world to deal with databases and services, or do non-deterministic things like generating random UUIDs, but a lot more of your program can be deterministic than you might realize. The more you explore programming with functions, the more these patterns will reveal themselves. Most of the time, you're at most one simple refactor away from achieving determinism: separate how you get data (non-deterministic) from what you do with it (deterministic).

6.2.3 Functions let you work locally

Most Java code is not built around functions. Methods constantly reference things outside of themselves – instance state, services, etc. Often the hardest part of understanding what a piece of code does has nothing to do with its actual logic. The hard part is building up a mental map of how that logic should behave given the *state of the everything going on outside of the method*.

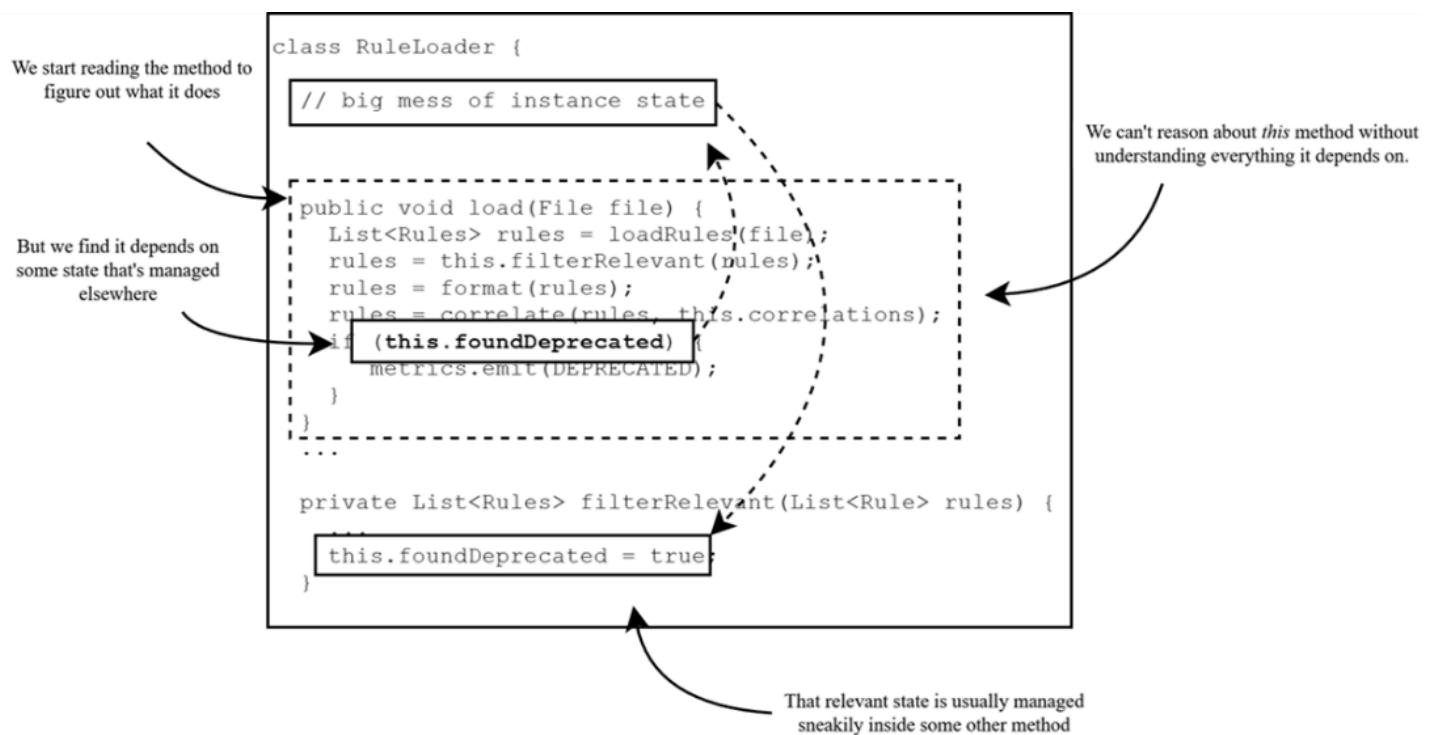


Figure 6.2 The chaotic jumping around required when reading a lot of Java code

This is the other thing that makes functions so special. They're isolated. Everything they need is supplied as an argument. Everything they "do" is returned as an output. You can drop into any function in any code base and understand what it does (it will map inputs to outputs!).

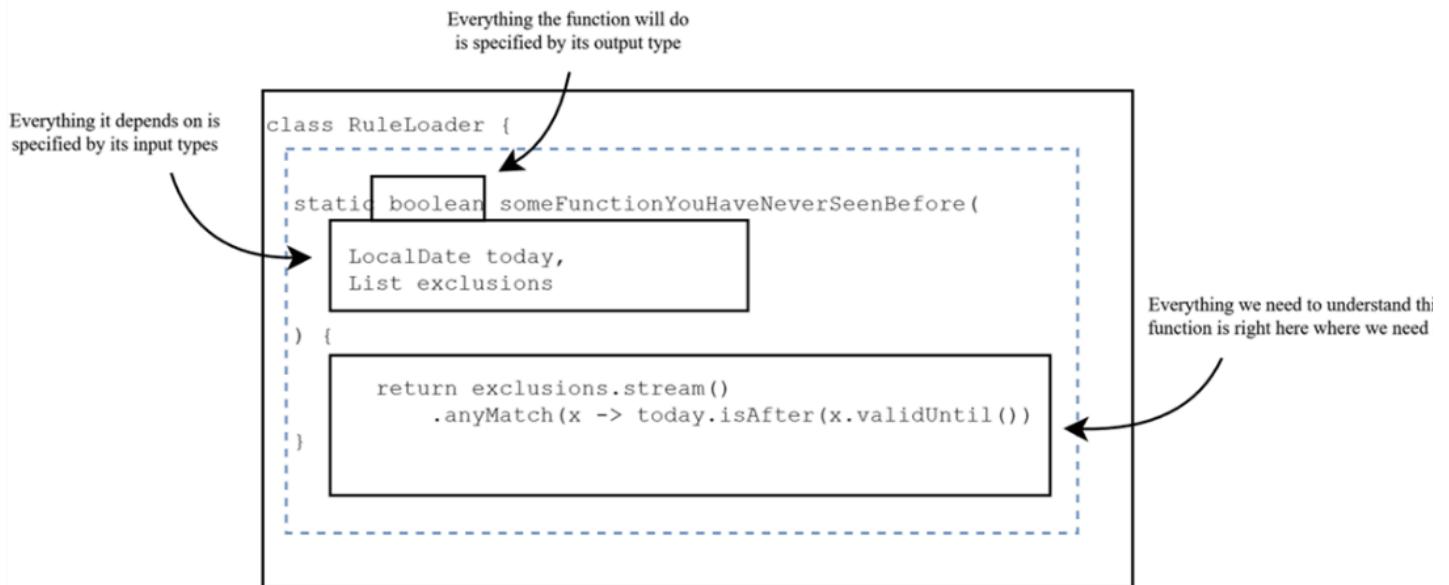


Figure 6.3 Everything we need to know is right there in the inputs and outputs

This local reasoning extends to your test suite as well. To understand a function, you only need to look at its inputs and outputs. Test “setup,” if there is any, becomes about crafting those inputs. No complicated setup and teardown, mocking, or orchestrating subtle state changes. Tests get reduced down to asserting that known inputs lead to expected outputs.

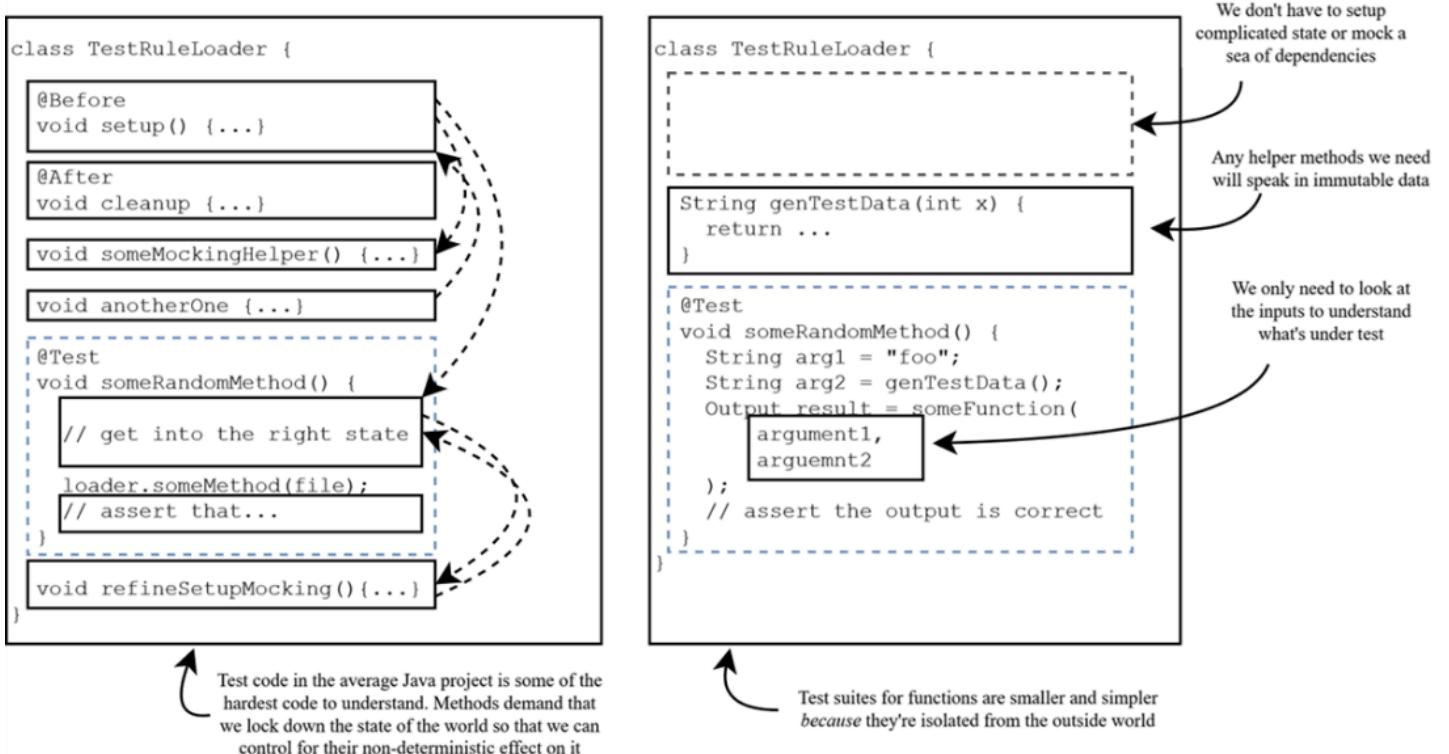


Figure 6.4 Comparing the patterns between testing methods and functions

6.2.4 Functions are tables of immutable data in disguise

Functions give us omniscience. Their determinism means we know every possible thing that the function *can* or *will* ever do. Every output is pre-determined.

In fact, because everything is pre-determined, we really don't even need the *function* part! We only need the *mapping* it describes. Every function is (conceptually speaking) just a big lookup table of inputs and outputs.

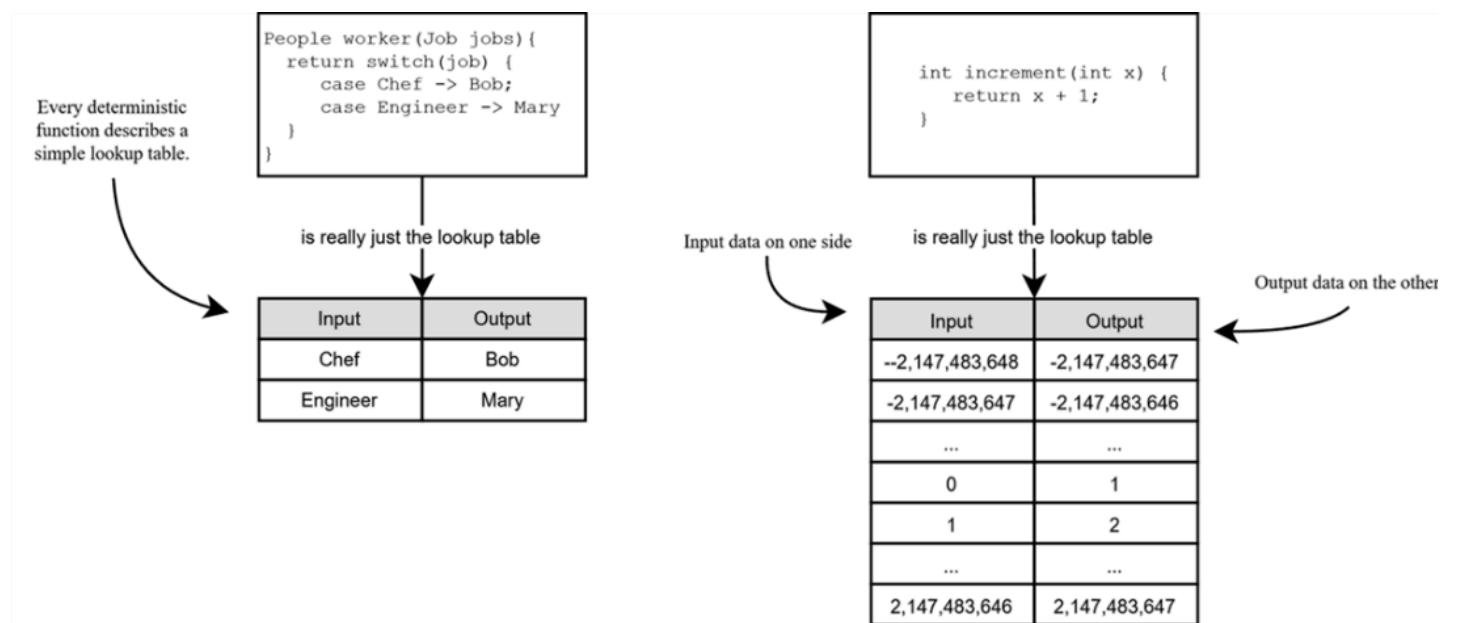


Figure 6.5 Behind every function is a lookup table of its inputs and outputs

You could imagine constructing this lookup table by looping over all of the possible inputs and storing them alongside their computed output.

Listing 6.16 Pre-computing the output for every possible input

```
static int increment(int x) { #A
    return x + 1; #A
} #A

Map<Integer, Integer> ANSWERS = new HashMap<>(); #B
for (int i = Integer.MIN_VALUE; i < Integer.MAX_VALUE-1; i++) { #B
    ANSWERS.put(i, increment(i)); #B
}
```

#A Every output of this function is pre-determined.

#B Conceptually speaking (don't actually do this unless you want an OutOfMemoryError), we could store every input and output in one big lookup table. Our code doesn't need the function – only the mapping it describes!

Any function *execution* can be swapped for a *look up* of a pre-computed answer without any change in our program's behavior.

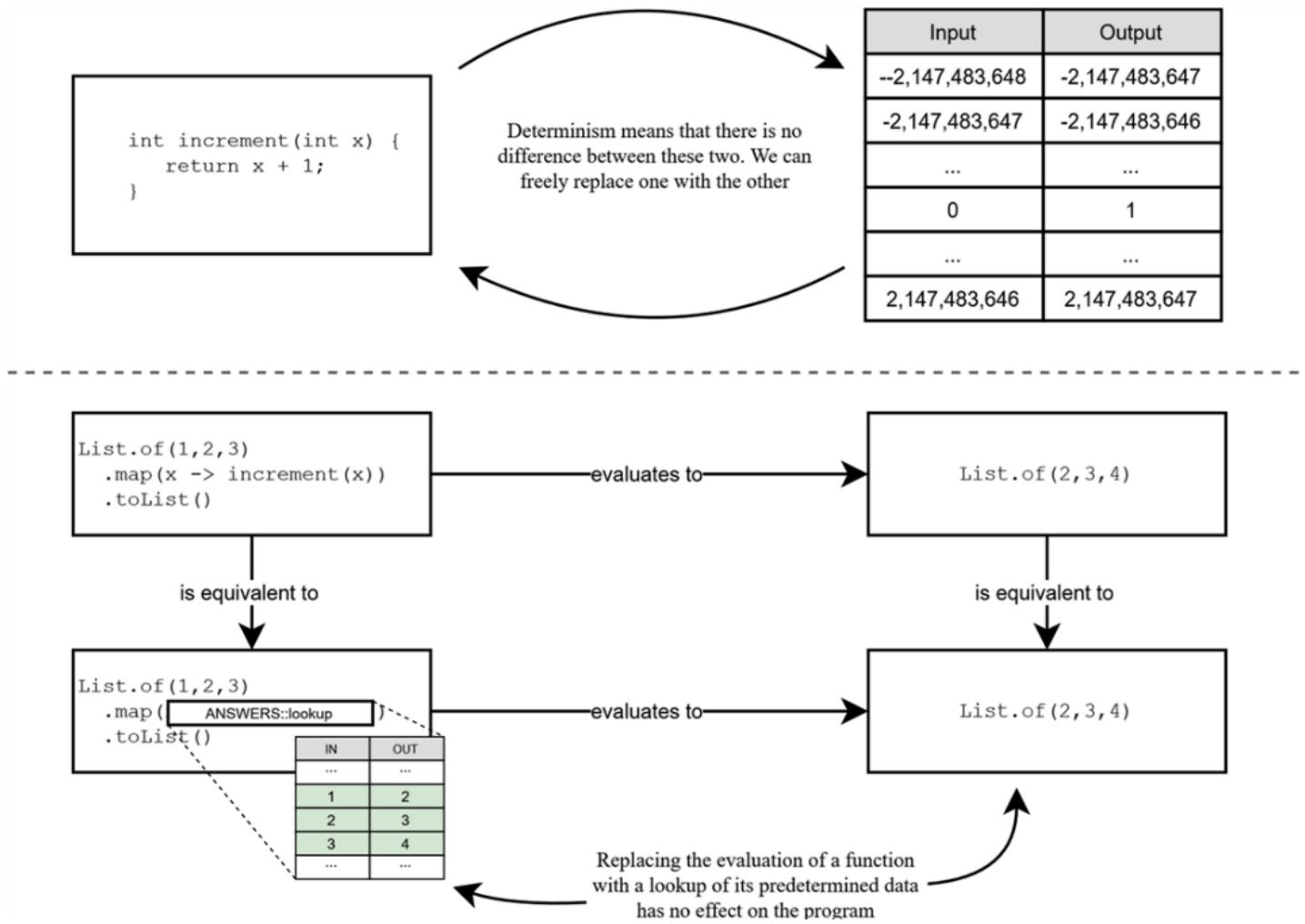


Figure 6.6 The relationship between immutable data and functions

This ability to replace functions with their values without changing the program is the “referential transparency” idea mentioned earlier. If functions are mappings from inputs to outputs, and those mapping never change because they’re deterministic, then it doesn’t matter if we execute the function, or lookup the answer in some table, or skip all of the computation and replace everything with the pre-determined answer directly. All of those lead to equivalent programs.

All of these produce equivalent programs

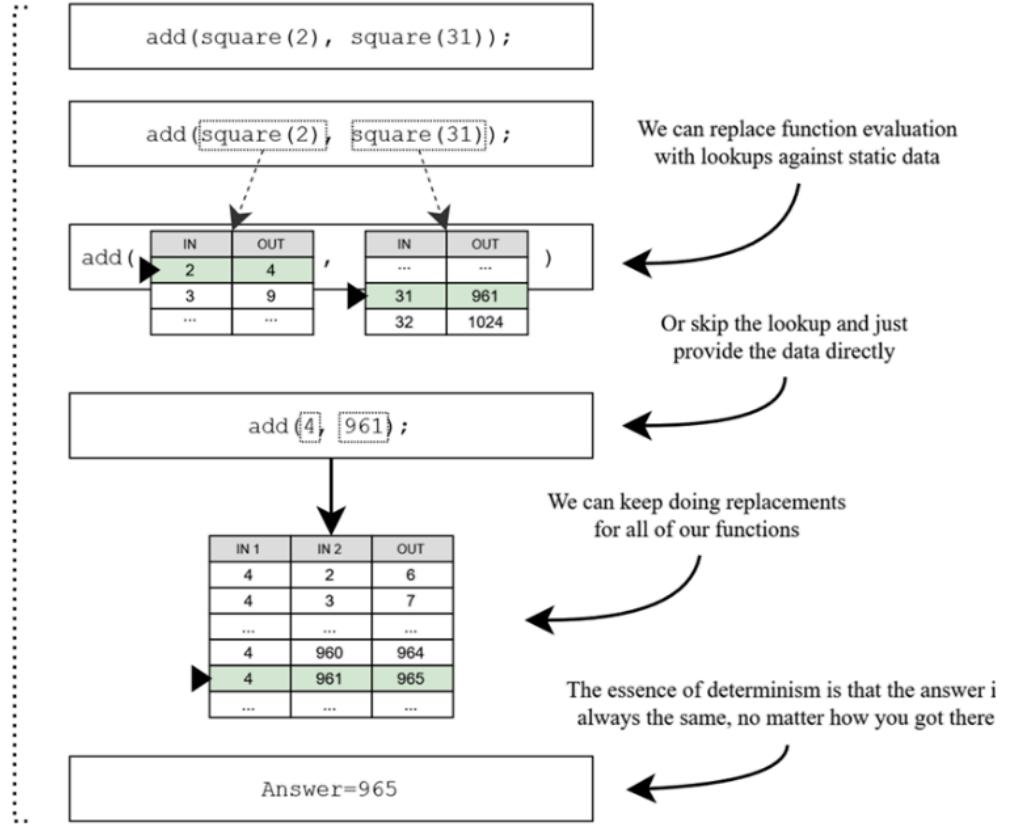


Figure 6.7 Deterministic functions can be replaced with data without changing your program's result

Being able to replace computation with plain data is hugely useful while testing and debugging our programs. When we only depend on functions, their determinism enables us to skip their execution with absolute confidence that we won't be missing out on any secret state updates. A function can always be replaced with the value it computes.

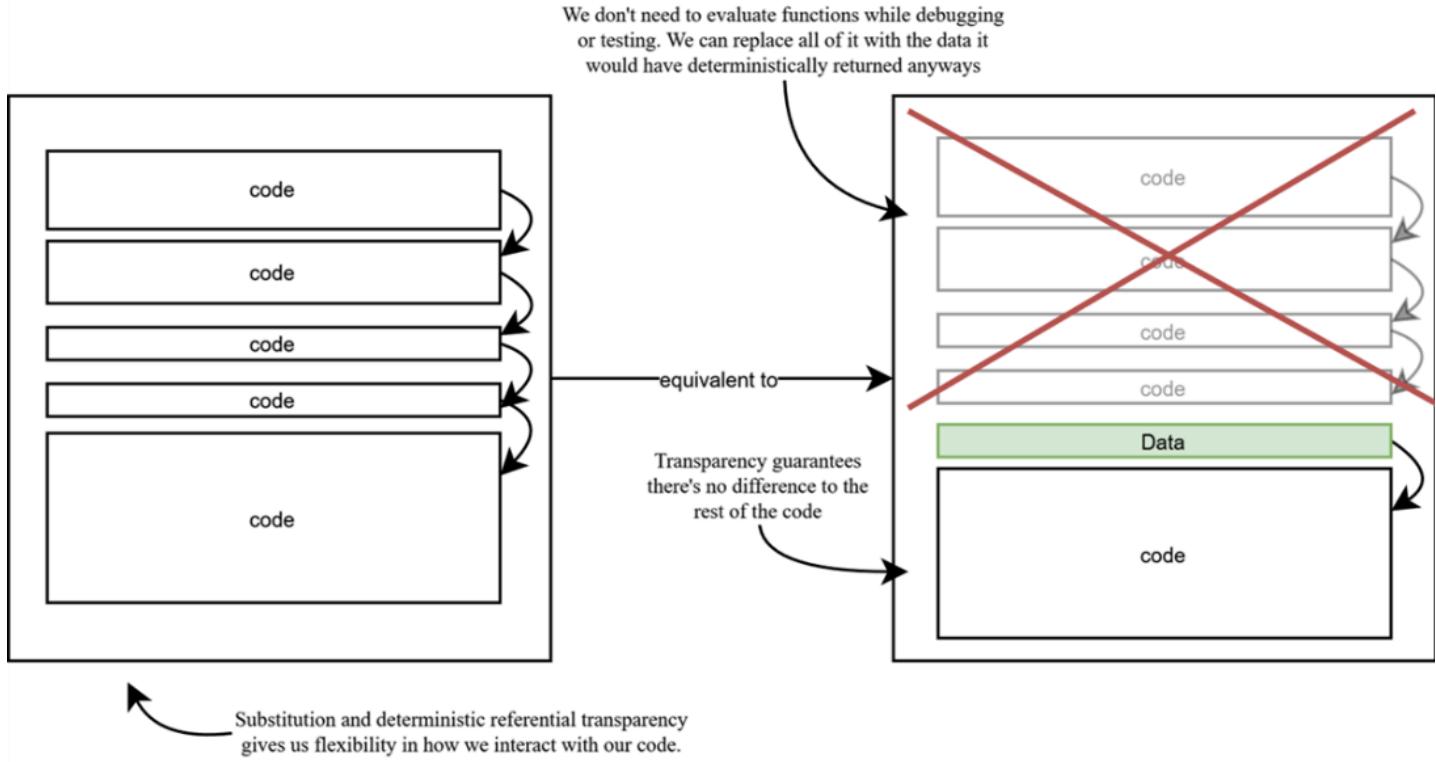


Figure 6.8 The flexibility that replacing computation with data brings

6.2.5 Functions let us model relationships

At first, you'll probably think about functions through a very "Java" lens. They'll be mostly in the role of "doers" like methods.

But there's another way of looking at functions. Their deterministic nature makes the line between where functions end and data begins blurry. If you squint, you can view the function *itself* as being a kind of data – one that describes a relationship between sets of values.

It's less esoteric than it initially sounds.

In our domain, grace period is an example of exactly this kind of relationship.

Table 6.1 The requirements for Grace Period

B0	The customer shall have a Grace Period
B1	Customers in good standing receive a 60-day grace period
B2	Customers in acceptable standing receive a 30-day grace period
B3	Customers in poor standing must pay by end of month

Grace period gives customers a buffer of extra days after their due date before we officially consider their payment late. That's a pretty straight forward idea, but what should it look like? How would we design it as a deterministic function? What is it that CustomerRating deterministically maps to?

Listing 6.17 Designing gracePeriod

```
static ??? gracePeriod(CustomerRating rating) { #A  
    ???  
}
```

#A GracePeriod maps the customer's rating to some kind of offset measured in days. How do we capture that?

We're good data-oriented programmers, so we might quickly run through our "what does it denote?" exercise and try plugging in a few data types. We could try making it a function from `CustomerRating` to `Integer`, or since we're talking about days, maybe `NonNegativeInt` (since negative days doesn't make sense). But that still has the problem of not saying what those integers *mean* – So, maybe we introduce a `Days` data type?

Listing 6.18 Does throwing more data at the problem help?

```
Days gracePeriod(CustomerRating rating) {  
    return switch(rating) {  
        case CustomerRating.GOOD -> new Days(60);  
        case CustomerRating.ACCEPTABLE -> new Days(30);  
        case CustomerRating.POOR -> ??? #A  
    }  
}
```

#A Hmm. What goes here? This is very different from the other cases

But all of these get stuck at the same spot. There's not a satisfying *single* data type that captures what grace period means.

The problem is that third requirement, B3, "Customers in poor standing must pay by end of month." It's different from the others. To know what "end of the month" means, you have to know the date against which you're applying this grace period.

One way to approach this would be to pass that additional context into our function.

Listing 6.19 expanding the surface area of our function

```
Days gracePeriod(CustomerRating rating, LocalDate dueDate) { #A  
    ...  
}
```

#A Supplying the "missing" context our function needed in order to handle its final case

But there's another way of looking at this. Grace Period expresses a *relationship* between an input date and an output date. It's not about a fixed offset of "days" – that's our modeling in code trying to push the meaning in a convenient direction. If we really listen to what Grace Period means, it's describing a *relationship* between two dates.

In other words: *a function*.

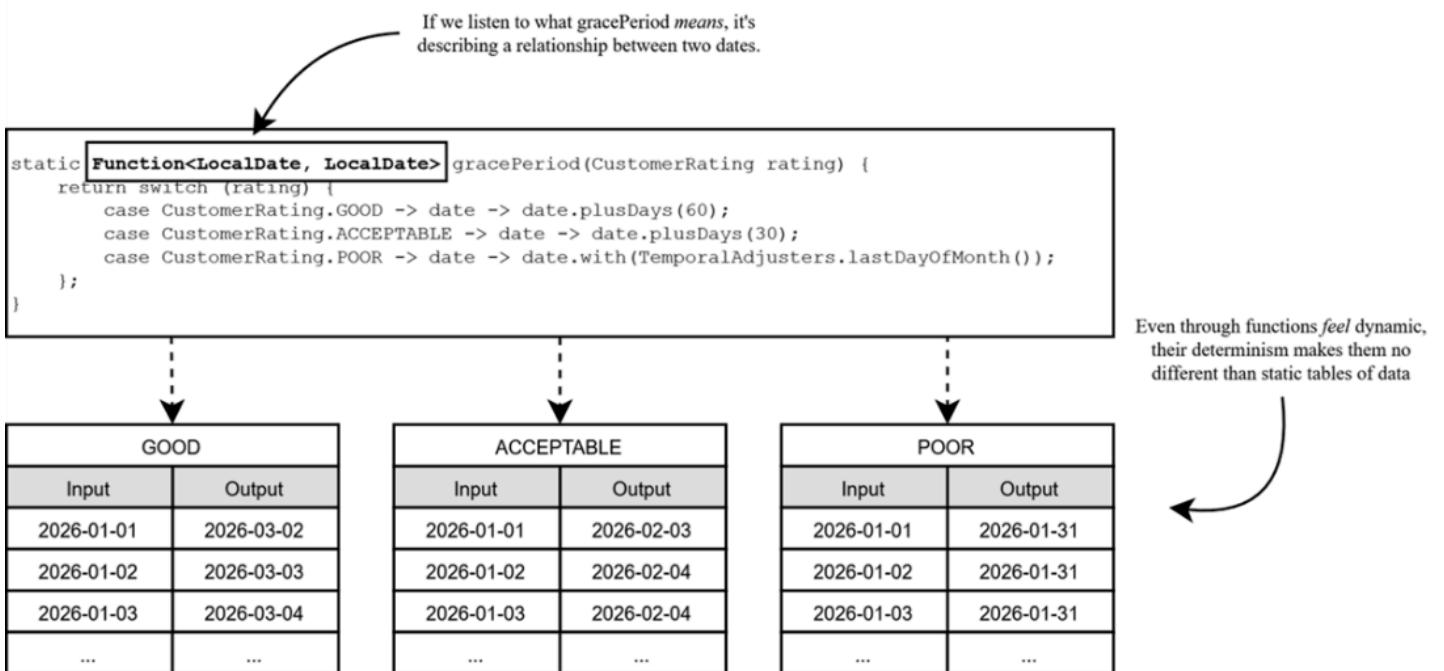


Figure 6.9 Returning a function as data

This type signature is every bit as deterministic as any “normal” data-returning function. It still deterministically maps inputs to outputs. The only difference is that the output is itself another deterministic function. Despite this, it retains all of its deterministic properties. The same inputs will always yield the same outputs.

Listing 6.20 Same inputs; Same outputs

```
gracePeriod(CustomerRating.GOOD).apply(dueDate) #A
```

#A The same inputs will always give the same outputs

This captures exactly what grace period means. It isn’t a fixed amount or some scalar value. Grace Period is a *function* from one point in time to another. It maps the customer’s rating to some kind of temporal offset.

The only downside of this precision is that defining that function is kind of noisy. Luckily, Java comes with its own built in function type that handles this exact mapping. Rather than roll our own, we can just use Java’s and simplify the type signature.

Listing 6.21 Using Java’s built-in function that maps time → time

```

static TemporalAdjuster gracePeriod(CustomerRating rating) {
    return switch(rating) {
        case CustomerRating.GOOD -> date -> date.plus(60, DAYS);
        case CustomerRating.ACCEPTABLE -> date -> date.plus(30, DAYS);
        case CustomerRating.POOR -> TemporalAdjusters.lastDayOfMonth();
    };
}

```

The reward for this semantic precision is that we get code that reads exactly like its requirements.

Listing 6.22 Code that reads just like the requirements

```
// The requirement
// A0 - The system shall charge late fees to customers with
// open past due invoices as of the Evaluation Date plus the
// customer's Grace Period

// The Code:
static boolean isPastDue(...) {
    evaluationDate.isAfter(invoice.dueDate().with(gracePeriod(rating)));
}
```

BUT WHAT IF WE...

You might be wondering why we're stressing over the semantics of grace period at all. Why not skip all that and just apply the business rules directly to the date?

Say, something like this.

Listing 6.23 An alternative implementation

```
static LocalDate mustHavePaidBy(          #A
    Invoice invoice,
    CustomerRating rating) {
    return switch(rating) {
        case CustomerRating.GOOD ->
            invoice.dueDate().plusDays(60);  #B
        // etc...
    };
}
static boolean isPastDue(...) {
    return today.isAfter(mustHavePaidBy(invoice, rating));  #C
}
```

#A We have to jiggle the naming a bit, since "grace period" isn't really about a specific date, it's something added to an existing one.

#B Applying the grace Period rules directly to the date

#C And this is fine, too! This approach is also delightfully clear!

Honestly? This is totally fine, too. I would have zero complaints if this popped up in a code review. We're well into the world of personal opinion and stylistic choices.

There is really only one thing that pulls me towards the other implementation. I'm always trying to get the stuff I know about the domain out of my head and into the code. *Especially* its language. "Grace Period" is a core idea in our domain. If we change or rewrite it to make our implementation easier, something subtle is lost along the way. Making gracePeriod its own function lets us highlight the domain concept as something important. We gave it a name, and show what that name means, because that name matters to the domain.

6.2.6 One last note on all this deterministic business

Watch out for zealotry. "Basically deterministic" is generally just as good as "actually deterministic." You're going to encounter lots of places in Java that require bridging the gap between identity objects and data. These will always come with "theoretical" non-determinism thanks to their mixed nature. We have plenty in our design so far.

Listing 6.24 The PastDue data type (and its dependency on an identity object)

```
record PastDue(Invoice invoice) {} #A
```

#A Invoice is a mutable identity object

In a turbo-pedantic technical sense, any function that uses this data type won't be truly deterministic. The "same" input could lead to different outputs if that mutable reference gets secretly updated.

But ignore all of that. The conditions where that would be an issue are so specific and rare that they're not worth fretting about. As long as we program as though these inputs *are* data, we'll be able to reap all of the benefits that deterministic code brings.

6.3 Organizing the code around determinism

Deterministic functions give a simple but powerful heuristic for organizing a program. We can use it to draw a dividing line. Deterministic things on one side; non-deterministic things on the other. This idea, though simple, is instructive.

The best "design patterns" are the ones that nudge us in the right direction. If deterministic functions are going to stay deterministic, there are some invariants about the design of our codebase that must always hold.

For one, deterministic things can only depend on other deterministic things. Obviously, if we start talking to a database or generating random numbers, we're no longer deterministic. So, a natural "core" starts emerging in the codebase. Inside of it, everything is deterministic. It can't reach outside of itself because that would destroy its determinism. And this simple demarcation causes a very useful architectural property to emerge: the dependency arrows only flow in one direction.

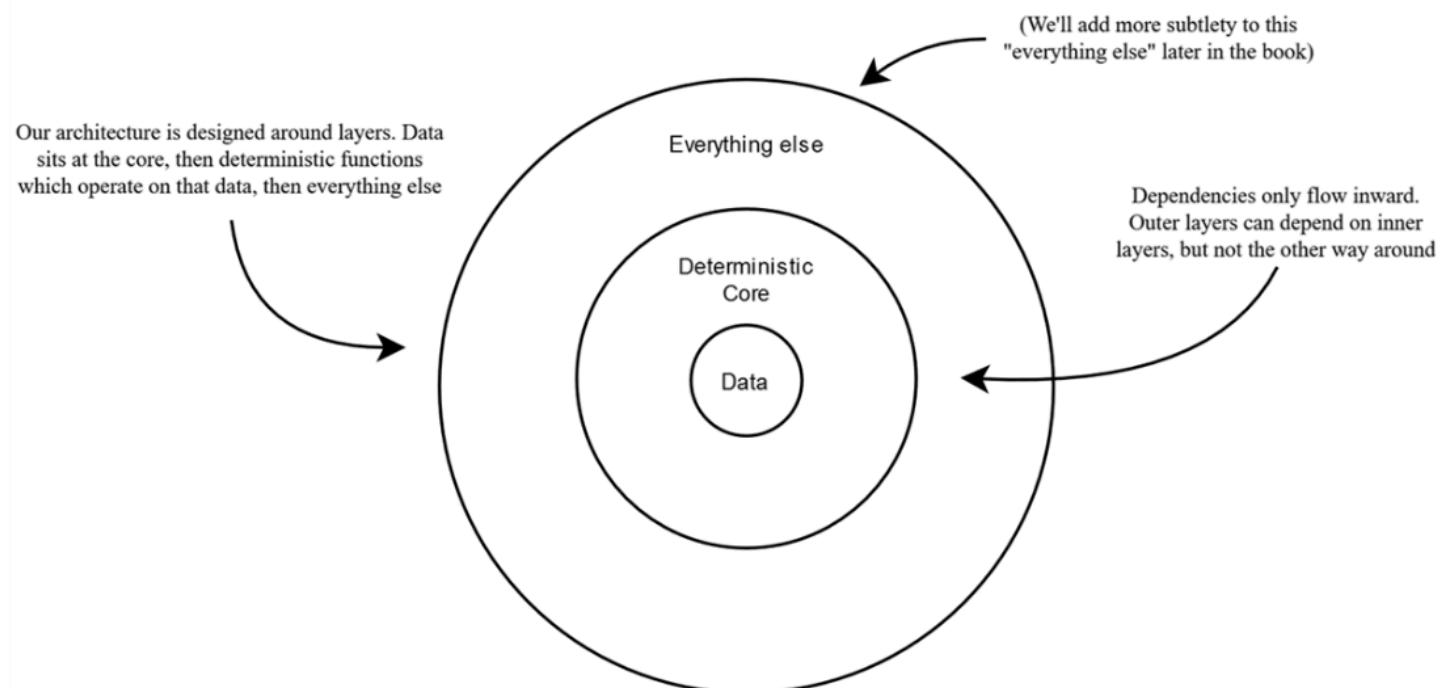


Figure 6.10 Building around a deterministic core

This picture is a very popular one and goes by many different names. The most common is probably “functional core; imperative shell.” However, this is another place where I’ll shy away from paradigm specifics (it’s very easy to get wrapped up in holy wars), and just focus on the property we want from this division: determinism.

This is the criteria by which we’ll divide up our implementation.

Using this, we can split the behaviors into two buckets.

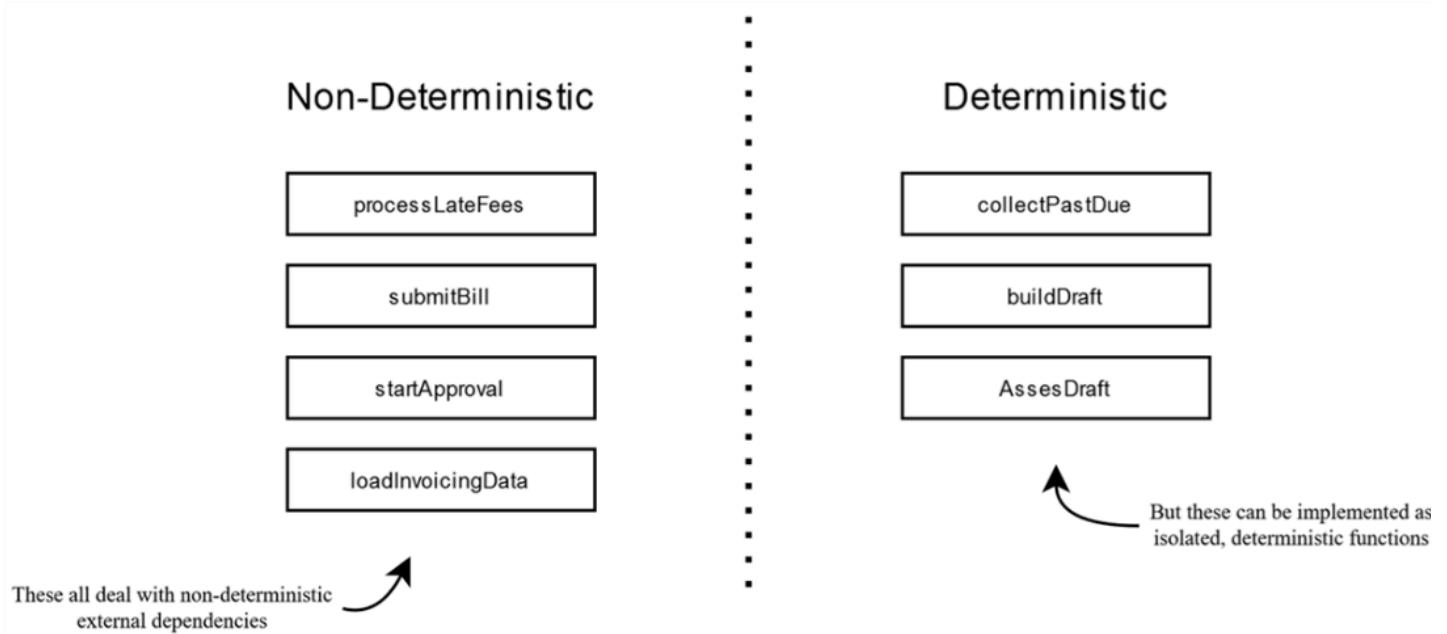


Figure 6.11 Dividing the world in two

6.4 Implementing the deterministic core

Now we’ll finally get our hands dirty and write some code. We’ll start with the deterministic core and work our way outward.

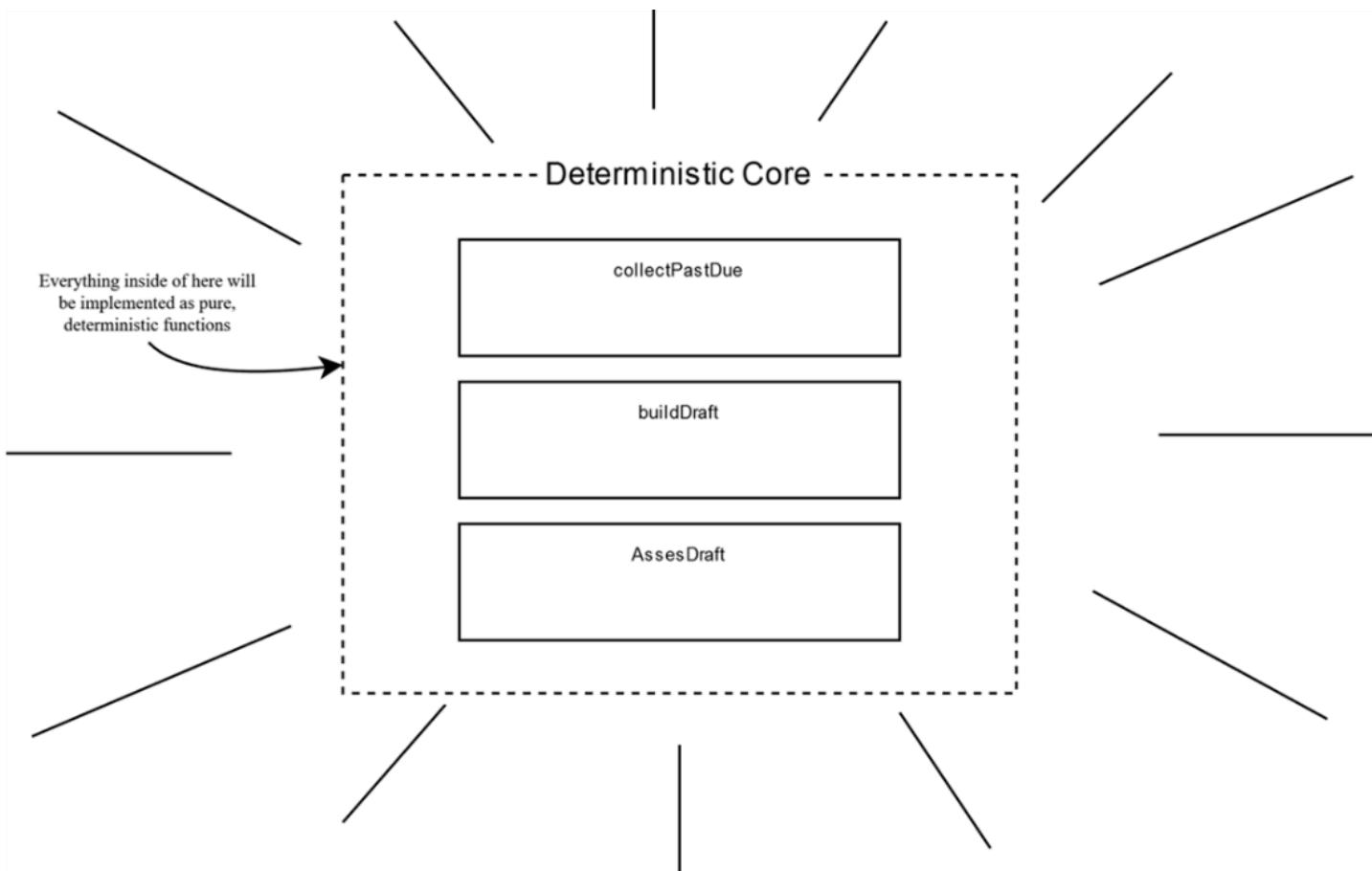


Figure 6.12 The deterministic core

The first thing to decide is where to put all of this special deterministic code. Ultimately, it doesn't matter. It could all go in the main Service class. Or maybe in its own package, or split across a suite of classes – the specifics matter less than the convention.

It's nice when you can look in the same spot for every feature to find where the deterministic business logic lives. For the purposes of this book, and making things explicit, we'll put all the deterministic stuff in a class literally, and uncreatively called "Core."

Listing 6.25 Imagine a file structure kind of like this

```
com.dop.invoicing
 |- latefees
 |   |- Core #A
 |   |- Service
 |   |- Types
```

#A This is where we'll put all of our deterministic code

This is not gospel or how you must do things in order to be "Data-Oriented." It's just something to think about. Putting things in their own class creates a useful boundary between the deterministic and non-deterministic parts of our code.

Ok. Let's get started!

6.4.1 Implementing collectPastDue

Here's where we left off in the last chapter.

Listing 6.26 Our model from the previous chapter

```
public static List<PastDue> collectPastDue(
    EnrichedCustomer customer,
    LocalDate today,
    List<Invoice> invoices) {
    // Implement me!
}
```

The type signature tells us everything this function needs to do. In fact, it tells us so much that we can really just sort of auto-pilot our way to an implementation by doing what they say. The data types have made our function "small." There are only a few implementations that could satisfy this type signature.

Listing 6.27 the finished implementation

```
public static List<PastDue> collectPastDue(
    EnrichedCustomer customer,
    LocalDate today,
    List<Invoice> invoices) {
    return invoices.stream() #A
        .filter(invoice -> ???)
        .map(PastDue::new) #A
        .toList(); #A
}
```

#A The shape of the implementation follows what the types say

The only question not answered by our data types is how to implement the filtering logic. Here's a refresher on the requirements.

Table 6.2 Requirements for how we determine past due

A0	The system shall charge late fees to customers with open past due invoices as of the Evaluation Date plus the customer's Grace Period
----	---

Whipping together a small helper method is pretty easy.

Listing 6.28 Checking if an Invoice is past due

```
static boolean isPastDue(
    Invoice invoice,
    CustomerRating rating,
    LocalDate today) {
    return invoice.getInvoiceType().equals(STANDARD) #A
        && invoice.getStatus().equals(OPEN) #B
        && today.isAfter(invoice.getDueDate().with(gracePeriod(rating))); #C
}
```

#A (this is part of our ongoing fuzzy barrier with the outside world. We have to make sure these Invoices are the right type due to database details beyond our control)

#B Now our core constraints. Invoices must still be open (unpaid)

#C and the current date must be after their due date and gracePeriod

The only interesting part is `gracePeriod`. If you recall, we sketched that out back in Section 6.21. It looks like this:

Listing 6.29 Grace Period (we already figured this one out in Listing 6.21)

```
static TemporalAdjuster gracePeriod(CustomerRating rating) {  
    return switch (rating) {  
        case CustomerRating.GOOD -> date -> date.plus(60, DAYS);  
        case CustomerRating.ACCEPTABLE -> date -> date.plus(30, DAYS);  
        case CustomerRating.POOR -> TemporalAdjusters.lastDayOfMonth();  
    };  
}
```

However, there's one more thing that's worth talking about with this implementation that we didn't cover in the previous section.

Do we need this method at all?

BUT WHAT IF WE...

A popular recommendation in some data-oriented circles is to "just use Maps." If it *can* be expressed as a data structure, then it *should* be expressed as a data structure.

This can be an alluring idea. After all, functions being deterministic means that there's no difference between computing a value and looking it up in a map. So, why not skip the computation wherever we can?

Listing 6.30 Modeling Grace Period as a map of data

```
static Map<CustomerRating, TemporalAdjuster> gracePeriod      #A  
= Map.of(  
    CustomerRating.GOOD, date -> date.plus(60, DAYS);  
    CustomerRating.ACCEPTABLE, date -> date.plus(30, DAYS);  
    CustomerRating.POOR, TemporalAdjusters.lastDayOfMonth();  
)
```

#A Now the business logic is just another piece of data!

Looking up the answer rather than computing it feels quite elegant.

Listing 6.31 Using the map of GracePeriods

```
currentDate.isAfter(  
    invoice.invoiceDate()  
    .with(gracePeriod.get(customer.rating()))) #A
```

#A pretty clean!

However, this change does something dangerous. It traded compile time *guarantees* for runtime assumptions. We *assume* that the map is total and every case is handled. Even worse, we assume that it will *always* be handled even as the code evolves.

But we cannot control the future.

Maps allow keys to be missing. If we're going to do our job well, we have to handle that possibility. And this lands us back into the problem we've been exploring over and over

in this book: illegal states that only exist due to our representation.

Listing 6.32 Defending against future unknowns

```
currentDate.isAfter(  
    invoice.invoiceDate()  
    .with(gracePeriodAdjustments  
        .getOrDefault(customer.rating(), ???) #A
```

#A What goes here? What would make a good default?

That thrusts us into a strange position where we have to invent a solution to a problem that doesn't exist in our domain. There is no default grace period. So, maybe we ignore that and just let it throw a NPE if the key isn't found? But that's just a different flavor of the same invented problem. Compile time guarantees have been tuned into runtime defenses.

The very real-world problem here is that those Rating enums belong to an external API. Those can and do change – often silently and without warning. I've lost track of how often a "minor" version bump in a dependency came with program breaking changes. "Semantic versioning" is nice idea, but it offers no useful protection in practice.

This is what makes case statements such a powerful option. They *demand* exhaustiveness. In exchange, they give us guarantees that we'll be alerted of any change at compile time. A map of data, while elegant, cannot do that.

6.4.2 Implementing BuildDraft

The draft is where we start to assemble the pieces that make up a late fee.

Table 6.3 Requirements for creating the draft late fee

A0	The system shall charge late fees to customers with open past due invoices as of the Evaluation Date plus the customer's Grace Period	
A1	The fee shall be calculated as a percentage of the customer's total past due	
A2	Fees shall be computed in USD	
A3	The Due Date of the Fee Invoice shall be based on the customer's Payment Terms	

This one basically writes itself thanks to our effort during the design stage.

Listing 6.33 The implementation just follows the type signature

```
static LateFee<Draft> buildDraft(
    LocalDate today,
    EnrichedCustomer customer,
    List<PastDue> invoices) {
    return new LateFee<>(#A
        new Draft(), #B
        customer,
        ???, #C
        today,
        ???, #C
        invoices
    );
}
```

#A There aren't many other ways this function could look given the types!

#B Most of the implementation is just doing what the types tell us to do

#C These are the only choices we get to make in this function.

That's pretty neat, right? The types completely constrain what this function could be. This is a perfect example of code that writes itself. We don't really have a choice on what the implementation could be, because there's only one way that will compile.

The only thing we have to do is plug in some implementations that compute the fee and due dates.

We'll start with due date.

It just follows the meaning of the enumeration. "Net 60" becomes 60 days, and so on.

Listing 6.34 computing the due date

```
static LocalDate dueDate(LocalDate today, PaymentTerms terms) {
    return switch (terms) {
        case PaymentTerms.NET_30 -> today.plusDays(30);
        case PaymentTerms.NET_60 -> today.plusDays(60);
        case PaymentTerms.DUE_ON_RECEIPT -> today;
        case PaymentTerms.END_OF_MONTH -> today.with(lastDayOfMonth());
    };
}
```

You might notice a theme emerging. Well typed functions are short. In many cases, the very first thing we do in the body is start returning a result. We're just mapping inputs to outputs. That's all we need to do even for complex business logic.

Now let's compute the fee.

Table 6.4 Requirements for computing the late fee

A1		The fee shall be calculated as a percentage of the customer's total past due
A2		Fees shall be computed in USD

To compute the fee, we sum each invoice's line items and multiply it by a percentage. We could write all of that in one big method that does both tasks, but instead, let's

define is as two functions. One that does the summing, one that uses that result for computing the fee. Separating the two responsibilities lets our code read just like the requirements.

Listing 6.35 the implementation for computing the fee

```
static USD computeFee(List<PastDue> pastDue, Percent percentage) {  
    return computeTotal(pastDue).multiply(percentage.decimalValue()); #A  
}
```

#A "The fee shall be calculated as a percentage of the customer's total past due"

Pulling the summing logic into its own method also lets us focus on some interesting aspects. We're about to run into something *uncomfortable*.

Listing 6.36 Summing the line items and discovering something weird

```
static USD computeTotal(List<PastDue> invoices) {  
    return invoices.stream().map(PastDue::invoice) #A  
        .flatMap(x -> x.getLineItems().stream()) #B  
        .map(LineItem::charges()) #C  
        .map(USD::new) //hmm... #D  
        .reduce(USD.zero(), USD::add);  
}
```

#A Grabbing all of the invoices

#B And then each of their line items (while flattening into one list)

#C Then grab their charges and...

#D blindly convert the charges to USD on faith -- hang on.

That feels off.

Is it safe to justly blindly convert the currency on those line items to USD like that? What if it's *not* USD? Of course, it *should* be – everything in our domain *should* be – but the original modeling doesn't enforce it, and that necessitates either blind faith (like we've done here) or mounting more defenses (unexpected currencies in an invoicing system is *very bad*).

6.4.3 That really gross wrong feeling is a feature

That weird feeling is feedback from our design. The data model we created captures the fact that Invoices must all be in USD, but the rest of the code doesn't enforce it anywhere. Now we're faced with this weird mismatch deep down in the bowels of the application.

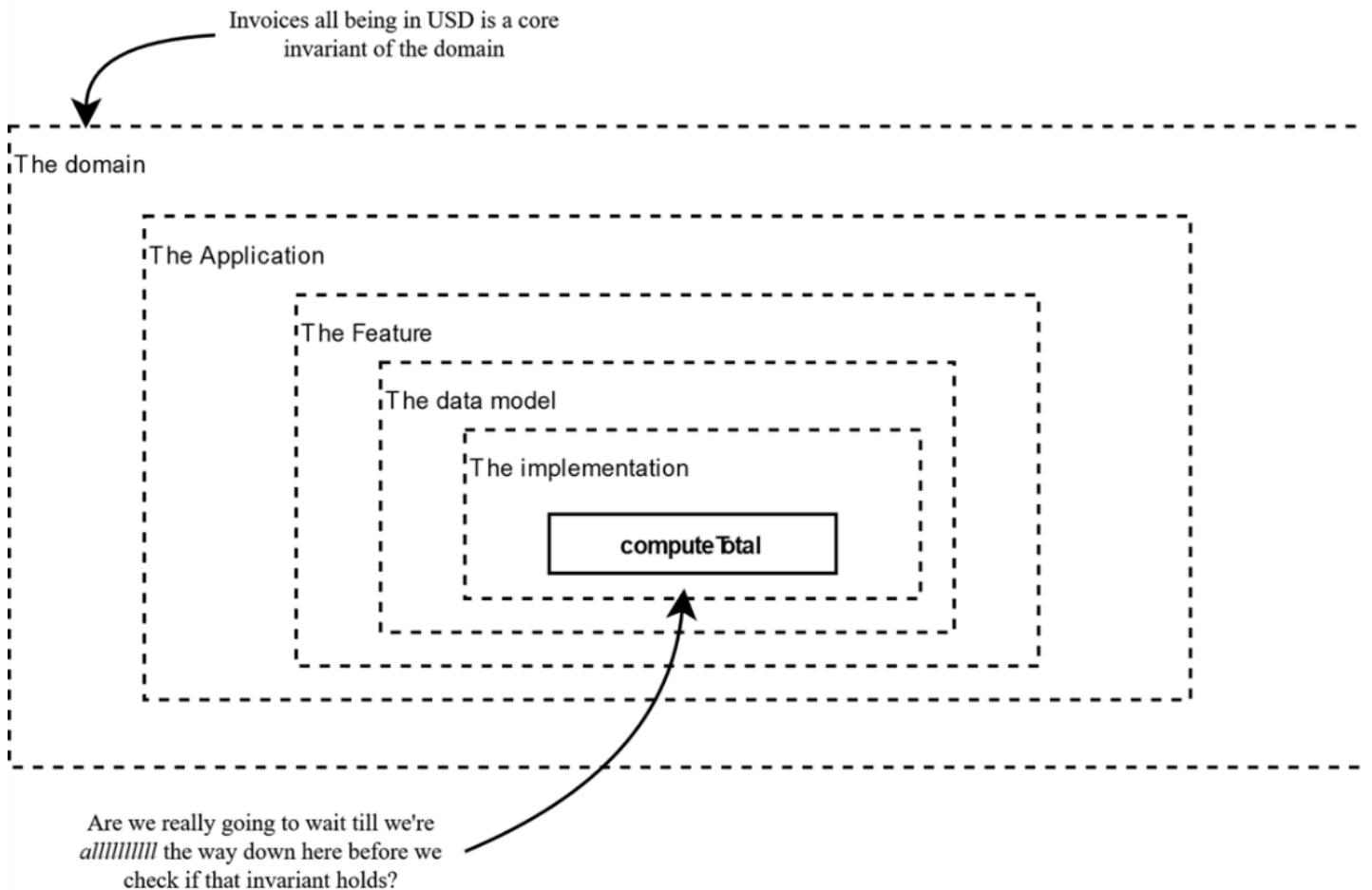


Figure 6.13 Is this where we should find out if the invariant holds?

This is one of my favorite things about encoding what we know into the type system. The USD data type doesn't *do* anything. It's just something we know about the domain. Yet, that's all it takes. The type acts like an assertion. This little type is enough to throw the design of our service as a whole into question.

You'll encounter this friction a lot as you start doing data-oriented programming. Listen to its feedback. Let it drive your codebase in a better direction.

But also, don't try to fix everything at once. Codebases move like large ships. We have to pick and choose our battles. Not every potential typing error represents something that needs fixed *right now*. For this case, where we generally expect everything to be in USD, it's enough to call out the gap, defend against it, and leave the fix for another day.

Something like this is usually a good option.

Listing 6.37 Calling out that we're trying to enforce something that might not hold

```
/**  
 * Attempts to convert the charges on the invoice to USD. #A  
 *  
 * In practice, we expect this never to be thrown, but  
 * that assumption relies on many systems all doing the  
 * right thing at all times.  
 *  
 * TODO: Validate invoices on the way into the system  
 * and harden data modeling.  
 * see issue: my-company-backlog.com/items/1234567 #B  
 */  
static USD unsafeGetChargesInUSD(LineItem lineItem)    #C  
    throws IllegalArgumentException {                  #D  
    if (!lineItem.getCurrency().getCurrencyCode().equals("USD")) {  
        throw new UnexpectedCurrencyException();  
    } else {  
        return new USD(lineItem.getCharges());  
    }  
}  
  
static USD computeTotal(List<PastDue> invoices) {  
    return invoices.stream().map(PastDue::invoice)  
        .flatMap(x -> x.getLineItems().stream())  
        .map(Core::unsafeGetChargesInUSD)    #E  
        .reduce(USD.zero(), USD::add);  
}
```

#A Lots of scary warning Javadoc

#B Putting an item in your backlog is a great idea. Get it prioritized!

#C A purposefully scary name calls attention to itself

#D We throw an unchecked primarily for the humans reading the code, and only secondarily as an absolute last-ditch defense against unexpected system states

#E Refactoring to use this new method.

Writing this code will always feel gross. It's weird that this random part of our service is the one place we actually check such an important invariant. But that's exactly what makes it powerful. You'll only deal with this grossness so many times before you get annoyed enough to go and verify these invariants at the front door.

FINISHING UP THE IMPLEMENTATION FOR buildDraft

Here's the final implementation for buildDraft.

Listing 6.38 The finished implementation for buildDraft

```
static LateFee<Draft> buildDraft(
    LocalDate today,
    EnrichedCustomer customer,
    List<PastDue> invoices) {
    return new LateFee<>(
        new Draft(),
        customer,
        computeFee(invoices, customer.feePercentage()),
        today,
        dueDate(today, customer.terms()),
        invoices
    );
}
```

6.4.4 Implementing Assess Draft

We're at the final and most complex function inside of the deterministic core. This is where we decide what to do with the draft we've created. As usual, here's a refresher on the requirements.

Table 6.5 Requirements for draft assessment

D0	The system shall not charge late fees for special cases
D1	Fees less than Minimum Due Threshold shall not be sent to the customer
D2	Fees greater than Maximum Due Threshold shall not be sent to the customer UNLESS the customer has been marked as Approved for Large Fees

Plugging in the rules from the requirements is easy enough. We're just seeing where the total falls relative to the allowed range. Too low and we can't bill. Too high and we defer to the state of the Approval. If we're in the sweet spot, we can bill without any further checks.

Listing 6.39 Our model from Chapter 5

```
BigDecimal total = draft.total().value() #A
if (total.compareTo(rules.minimumFeeThreshold()) < 0){           #B
    return new NotBillable(draft, new Reason("..."));
}
else if (total.compareTo(rules.maximumFeeThreshold()) > 0) { #B
    // we'll come back to this one
}
else {
    return new Billable(draft); #C
}
```

#A (We're doing a little rearranging in order to get things to fit on the page of a book).

#B Here are the border cases. Too low? can't bill. Too high? Well... we'll get to that below

#C Implicitly, if we make it here then we're in the goldilocks zone and can freely bill this customer

The hairy part of this function is that we won't necessarily *have* an approval. Most customers won't need them, so they're created lazily.

Listing 6.40 The Enriched Customer model

```
record EnrichedCustomer(  
    String id,  
    Address address,  
    Percent feePercentage,  
    PaymentTerms terms,  
    CustomerRating rating,  
    Optional<Approval> approval  #A  
){}  
}
```

#A Approval is optional because they're only created as needed

So, to finish implementing this function, we have to unfortunately dip our toes into the ongoing holy war that is the Optional data type in java.

6.4.5 Dealing with optional

Optionals bring out a lot of “thou shalt nots,” “You’re doing it wrongs,” and “uh... actually, you shoulds”. Camps get formed. Spears get sharpened.

The “trouble” with the data type is that you can interact with it with both imperatively and functionally. Often, the “functional” way gets assumed (usually by strong personalities) to be the “better” way. And they’re often right. There are tons of cases where the functional side of the API simplifies and clarifies our code.

Listing 6.41 Comparing imperative and functional interactions

```
static String imperativeExample(String thingId) {  
    Optional<String> maybeThing = this.tryToFindThing(thingId);  
    return maybeThing.isPresent()  #A  
        ? maybeThing.get().toUpperCase(); #B  
        : "Nothing found!"  
}  
  
static String functionalExample(String thingId) {  
    this.tryToFindThing(thingId)  #C  
        .map(String::toUpperCase)  #C  
        .orElse("Nothing Found!"); #C  
}
```

#A The imperative way involves manually checking the state of the optional

#B as well as manually grabbing its contents

#C The functional approach stays above the details and works declaratively

However, what often happens with these data types is that dogma sneaks in over time. The functional way becomes the correct way *because* it’s the functional way.

Younger me definitely fell into this trap. I’ve written tons of terrible and embarrassing code while cloaking myself against critique solely on the grounds of it being “functional” (and thus obviously better). For a while there, I forgot that programming is about communication with humans (and incidentally computers), and not just about expressing your prowess and adherence to the patterns of a certain paradigm.

It’s always worth questioning the dogma.

Is this approach

Listing 6.42 Handling missing Approvals functionally

```
draft.customer().approval().map(approval -> switch(approval.status()) {  
    case ApprovalStatus.APPROVED -> new Billable(draft);  
    case ApprovalStatus.PENDING -> new NotBillable(draft, ...);  
    case ApprovalStatus.DENIED -> new NotBillable(draft, ...);  
}).orElse(new NeedsApproval(draft));
```

Better than this one?

Listing 6.43 Handling missing approvals imperatively

```
draft.customer().approval().isEmpty()  
? new NeedsApproval(draft)  
: switch (draft.customer().approval().get().status()) {  
    case ApprovalStatus.APPROVED -> new Billable(draft);  
    case ApprovalStatus.PENDING -> new NotBillable(draft, ...);  
    case ApprovalStatus.DENIED -> new NotBillable(draft, ...);  
};  
}
```

Or are they just different?

The answer depends on how we're measuring "better."

There's an example that Larry McEnerney, director of the University of Chicago's Writing program, uses to demonstrate that most rules about writing stop being so when you consider them in context. It goes like this:

Which one of these is the better sentence?

1. The dog chased the cat
2. The cat was chased by the dog.

Most would argue that the first sentence is better. Arguments might be about active versus passive, or maybe brevity or whatever. But the real answer, Larry argues, is that it depends on what the reader cares about. These sentences *focus on different things*. The first is "about" the dog. The second is "about" the cat. Both carry the same information, but where they focus the reader's attention changes. If you care about the cat, then the second sentence, despite being passive and weaker, highlights the thing you care about.

That's how I think about Optionals. We design our interactions with them around what we want to focus our reader's attention on.

The imperative version in Listing 6.43, which many would argue is "doing it wrong," puts the reader's focus on the most interesting case. If we don't have an Approval, then we need to make one! That's lifecycle information worth highlighting. Everything else is just more case matching. The more "functional" option in Listing 6.42, shifts the focus of the code. It leaves that interesting empty case to an "orElse" tucked away at the very end.

This doesn't make one "more right" than the other (you might not even agree with my reading of the code). This is the art part of our craft. There are a lot of subjective decisions we make while trying to get our code to communicate our intentions. The

important part is that we drive our decisions from those intentions, and not out of blind purity to the way it “should” be done in one paradigm or another.

So, for this specific example, in this specific context, based on how it makes the code read, we’ll do the “wrong” thing, and go with the imperative option from listing 6.43.

Here’s the finished method.

Listing 6.44 The finished method

```
public static ReviewedFee assessDraft(  
    Rules rules,  
    LateFee<Draft> draft  
) {  
    BigDecimal total = draft.total().value()  
    if (total.compareTo(rules.minimumFeeThreshold()) < 0){  
        return new NotBillable(draft, new Reason("..."));  
    }  
    else if (total.compareTo(rules.maximumFeeThreshold()) > 0) {  
        return draft.customer().approval().isEmpty()  
            ? new NeedsApproval(draft)  
            : switch (draft.customer().approval().get().status()) {  
                case ApprovalStatus.APPROVED -> new Billable(draft);  
                case ApprovalStatus.PENDING -> new NotBillable(draft, ...);  
                case ApprovalStatus.DENIED -> new NotBillable(draft, ...);  
            } else {  
                return new Billable(draft);  
            }  
    }  
}
```

That’s a pretty good function, but there’s still something that bothers me about it. It’s visually noisy (being squashed in order to fit on a book’s page also isn’t helping). When you run across this one in the code, you’d have to pause and really stare at it to understand what’s going on.

Let’s see if we can make that better.

DESIGN PATTERNS AND MONADIC TYPES

We’ll have more to say about data types like Optional later in the book. The design of the humble Optional type houses many beautiful ideas that are worth study.

6.4.6 It’s always OK to introduce more types!

There’s something that has always bugged me about if/else chains where each case has a distinct semantic meaning. The “else” case doesn’t get to say what it is. It’s stuck being “not those other things.”

Listing 6.45 "I'm doing a not (tea or soda) run. You guys want anything?"

```
if (Condition A)      #A
else if (Condition B) #A
else
  ??? #B
```

#A We get to concretely show what these conditions are

#B But the else doesn't get to do the same! It's only not(A or B). How many other cases are in that "else"?

I feel like it subtly hides something important. That "else" means more than "not those other things." In our domain, it's the narrow goldilocks zone where we can bill without doing additional work. An Assessment is exactly *three* cases, not two cases plus anything that's "not those other two".

Data is cheap. It's OK to introduce new types. As many as you need. Anywhere you need them.

When presented with a clarity problem like this, we can reach into our increasingly common bag of tricks: separate the decisions our program makes from what we do with them.

Listing 6.46 Giving "not those other things" its own name

```
private enum Assessment {ABOVE_MAXIMUM, BELOW_MINIMUM, WITHIN_RANGE} #A

static Assessment assessTotal(USD total, Rules rules) { #B
  if (total.value().compareTo(rules.getMinimumFeeThreshold()) < 0){
    return Assessment.BELOW_MINIMUM;
  } else if (total.value().compareTo(rules.getMaximumFeeThreshold()) > 0) {
    return Assessment.ABOVE_THRESHOLD;
  } else {
    return Assessment.WITHIN_RANGE
  }
}
```

#A Introducing an enum to capture that an Assessment is exactly three things

#B And a new function that just handles the logic of assessing the total and returning its findings

Now we can use the result of our assessment in the main function.

Listing 6.47 Refactoring to make the 3 assessment cases explicit

```
public static ReviewedFee assessDraft(
    Entities.Rules rules,
    LateFee<Draft> draft
) {
    return switch (assessTotal(rules, draft.total())) {
        case WITHIN_RANGE -> new Billable(draft); #A
        case BELOW_MINIMUM -> new NotBillable(draft, new Reason("...")); #A
        case ABOVE_MAXIMUM -> draft.customer().approval().isEmpty() ? new NeedsApproval(draft)
            : switch (draft.customer().approval().get().status()) {
                case APPROVED -> new Billable(draft);
                case PENDING -> new NotBillable(draft, new Reason("..."));
                case DENIED -> new NotBillable(draft, new Reason("..."));
            };
    };
}
```

#A The three, and only three, assessment possibilities in our domain

These refactorings are another way of controlling what our reader focuses on. It's not something we need to (or should) do everywhere. This is just to show that it's OK to do if we think it'll help. Sometimes it's nice to be able to give each case an explicit name when what those names mean hold a lot of value in the domain.

6.4.7 Adding Support for updating LateFees

Before we move onto the rest of the implementation, we have to lay down a bit of support code for making "updates" (creating copies in a new state) to our `LateFee` type. They're included here just for completeness.

Listing 6.48 updating

```
record LateFee<State extends Lifecycle>(...) {

    public LateFee<Billed> markBilled(InvoiceId id) {
        return new LateFee<>(new Billed(id), customer, total, /*...*/);
    }

    public LateFee<Rejected> markNotBilled(Reason reason) {
        return new LateFee<>(new Rejected(reason), customer, total, /*...*/);
    }

    public LateFee<UnderReview> markAsBeingReviewed(String approvalId) {
        return new LateFee<>(new UnderReview(approvalId), customer, /*...*/);
    }
}
```

And with that, the deterministic core is done!

6.5 Implementing the non-deterministic shell

We're at the finish line.

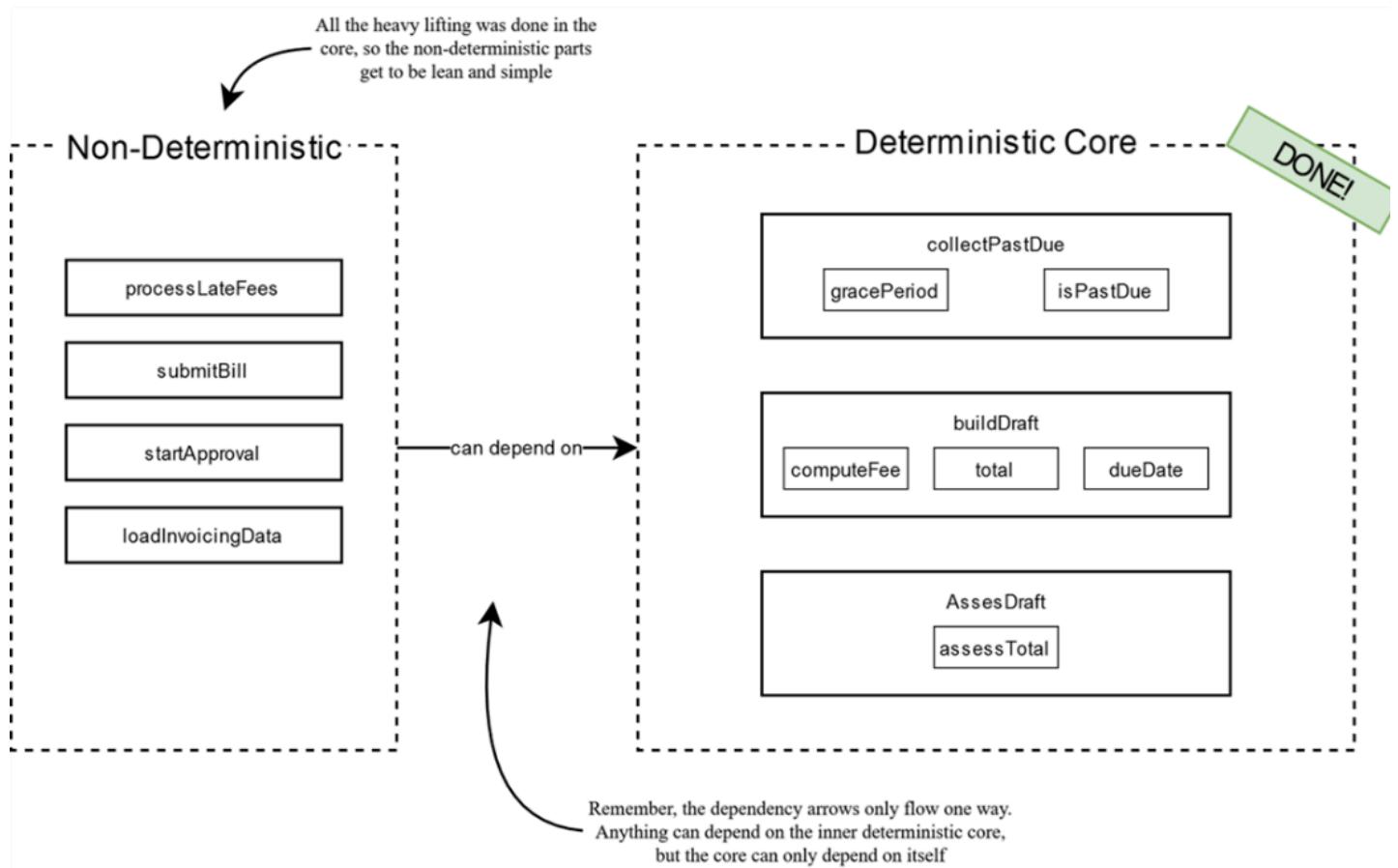


Figure 6.14 All that's left to implement

It's important to maintain discipline as we move out of the deterministic core.

Methods are free to do anything, but we should pretend like they're not. We should program as though they *are* functions. We'll listen to the type signature and let it guide our implementation. Those who read our code should find nothing surprising in body of a method. Just boring, predictable code that does exactly what it says it will and nothing more.

Let's start with the main method that ties all of this together.

This is where we'll offload work to the inner core. We can depend on it, but it cannot depend on us. This dependency constraint leads to common pattern that looks a bit like a sandwich.

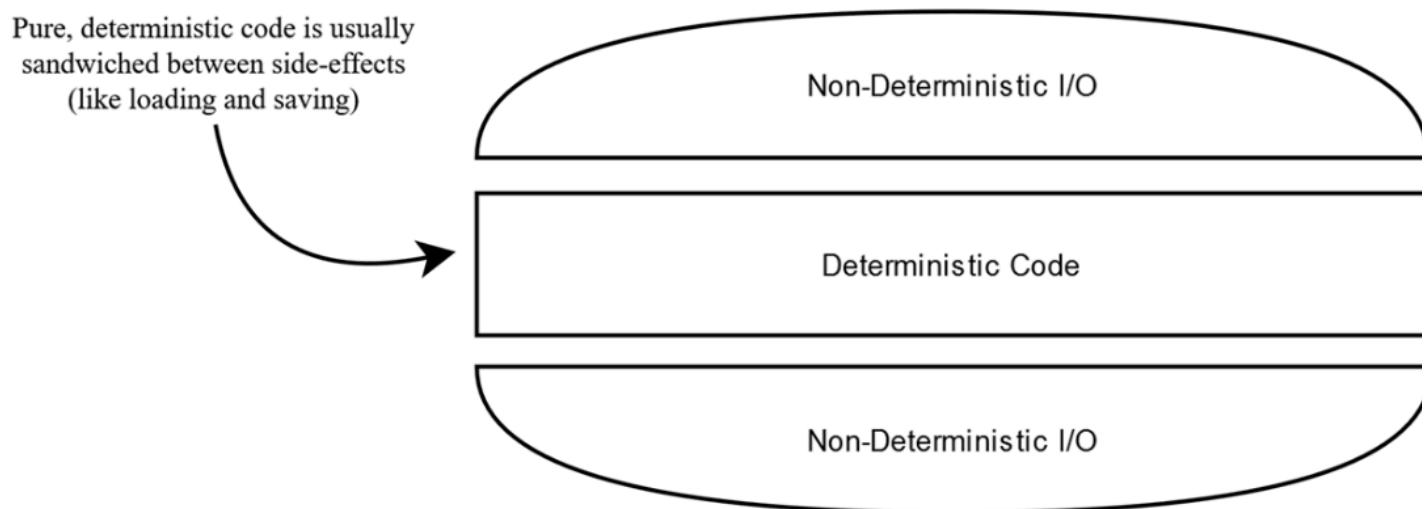


Figure 6.15. The sandwich

We do some non-deterministic work (like data loading), then feed those results into the deterministic core, then take those results and use them for more non-deterministic work (like saving the data).

It looks like this:

Listing 6.49 Roughing in the main method

```
public void processLatefees() {
    // ---- NON-DETERMINISTIC OUTER SHELL -----
    loadInvoicingData().forEach(data -> {
        // ---- DETERMINISTIC CORE -----
        LocalDate today = data.currentDate(); #A
        EnrichedCustomer customer = data.customer(); #B
        List<Invoice> invoices = data.invoices(); #B
        List<PastDue> pastDue = collectPastDue(customer, today, invoices); #B
        LateFee<Draft> draft = buildDraft(today, customer, pastDue); #B
        ReviewedFee reviewed = assessDraft(data.rules(), draft); #B

        // ---- NON-DETERMINISTIC OUTER SHELL -----
        // [more work here] #C
    });
}
```

#A We kick off the method by loading all the data we need

#B Then we feed that into our deterministic core. It works with the results of that non-deterministic I/O
#C Then feed those results into the next stage of side-effects and I/O

What does that next step look like? We just keep following the types. All we can do is pattern match.

Listing 6.50 We just do whatever the type system tells us

```
public void processLatefees() {  
    loadInvoicingData().forEach(data -> {  
        ...  
        ReviewedFee reviewed = Core.assessDraft(data.rules(), draft); #A  
        LateFee<? extends Lifecycle> latefee = switch (reviewed) { #B  
            case Billable b -> this.submitBill(b); #B  
            case NeedsApproval a -> this.startApproval(a); #B  
            case NotBillable nb -> nb.latefee().markNotBilled(nb.reason()); #B  
        };  
        this.save(latefee); #C  
    });  
}
```

#A ReviewedFee is a sum type, so the only thing we can do next is pattern match on it

#B What do we do here? Just follow the types! They tell us what cases to handle

#C The last thing we do is save all our work

The remaining methods, submitBill and startApproval, are similarly straight forward. Aside from a service call, they just do what their type signature tells them to do.

Listing 6.51 Implementing the billing and approval side-effects

```
LateFee<? extends Lifecycle> submitBill(Billable billable) {  
    BillingResponse response = this.billingApi.submit(/*...*/);  
    LateFee<Draft> draft = billable.latefee();  
    return switch (response.status()) {  
        case ACCEPTED ->  
            draft.markBilled(new InvoiceId(response.invoiceId()));  
        case REJECTED ->  
            draft.markNotBilled(new Reason(response.error()));  
    };  
}  
  
LateFee<UnderReview> startApproval(NeedsApproval needsApproval) { #A  
    Approval approval = this.approvalsApi.createApproval(/*...*/);  
    return needsApproval.latefee().markAsBeingReviewed(approval.id());  
}
```

#A Another good example of a method which is completely controlled by its data types

All that's left is saving our work. It comes with a few interesting things to consider.

6.5.1 Who owns the save logic?

How the heck do we save a data type like LateFee<? extends Lifecycle>? Where does that save logic go? Do we make our own repositories? Extend existing repositories to work with our data types? Start slapping Entity annotations all over our data model and give in to the ORM gods?

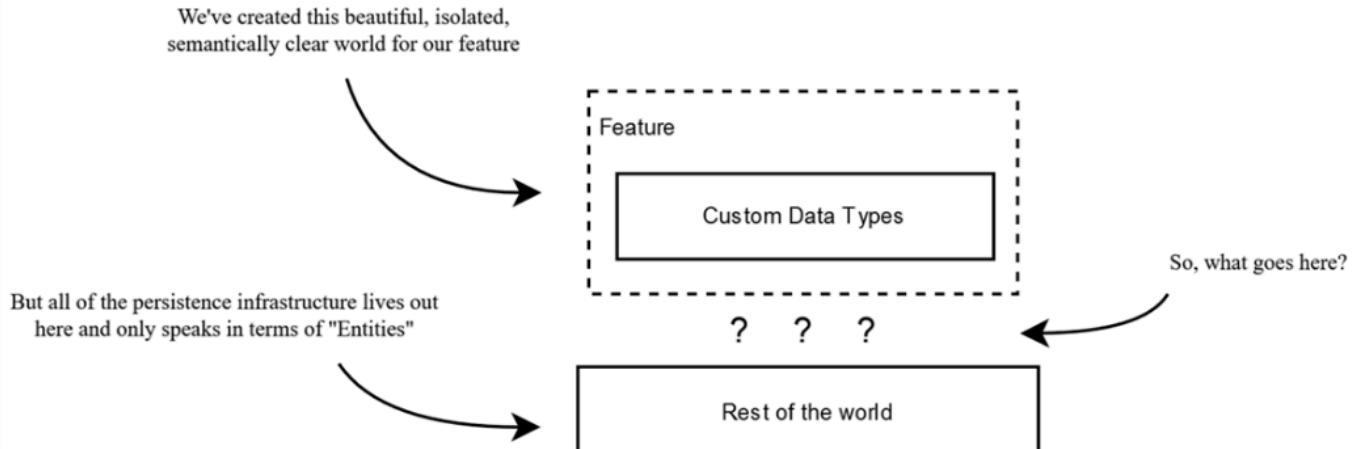


Figure 6.16 How do we get out of our world so we can save our work?

It's a hard call. It really depends on how much you want to start tugging your system as a whole in a new well-typed, data-oriented direction. There is surely more we'll want to do with a data type like `LateFee`. It embodies our domain in a way generic entity types like `Invoice` could never. However, the thing that usually dominates what we do is time.

There's just not enough of it.

So, the practical approach is let our world continue to be isolated from everything else. Rather than save our `LateFee` type directly, we'll transform it into a type that the rest of the existing world can understand.

6.5.2 Your Entities are my Data Transfer Objects (DTOs)

Everything outside of our little isolated world speaks `Invoice Entity`. Persistence is built around it. We can leverage this fact and make `Invoice` the exchange point between our two worlds. From our perspective, `Invoice` is not an Entity, it's just some Data Transfer Object.

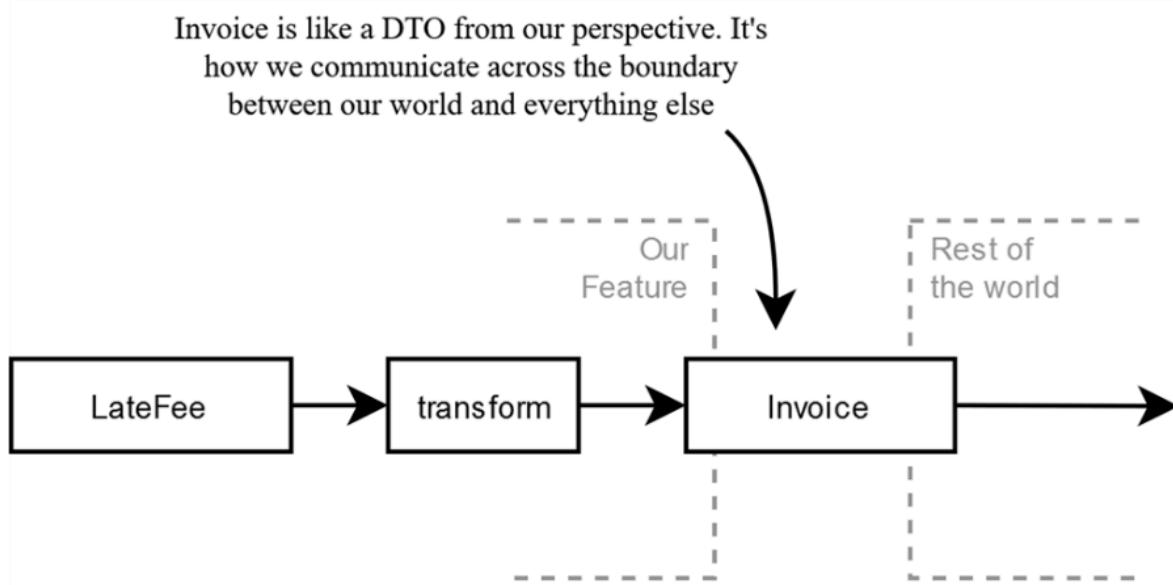


Figure 6.17 Treating the Invoice Entity as a DTO

All we have to do is deterministically map between the two.

Listing 6.52 Deterministically mapping between worlds

```
static Invoice toInvoice(LateFee<? extends Lifecycle> latefee) { #A
    Invoice invoice = new Invoice();
    invoice.setInvoiceId(switch (latefee.state()) { #B
        case Billed(var id) -> id.value();
        default -> Invoice.tempId(); #C
    });
    invoice.setCustomerId(latefee.customer().id().value());
    invoice.setLineItems(List.of(new LineItem(
        null,
        "Late Fee",
        latefee.total().value(),
        Currency.getInstance("USD")
    )));
    invoice.setStatus(InvoiceStatus.OPEN); #D
    invoice.setInvoiceDate(latefee.invoiceDate()); #D
    invoice.setDueDate(latefee.dueDate()); #D
    invoice.setInvoiceType(InvoiceType.LATEFEE); #D
    invoice.setAuditInfo(new AuditInfo(
        null,
        latefee.includedInFee().stream().map(PastDue::invoice).toList(), switch(latefee.state()) {
            case Rejected(var why) -> why.value();
            case UnderReview(String approvalId) -> approvalId;
            default -> null;
        }
    ));
    return invoice;
}
```

#A We define it as a static to cue that it's a function

#B We pattern match where needed

#C (This bit of weirdness was defined in chapter 5. We don't have "real" Ids until successfully billed)

#D Aside from the occasional pattern match, it's just some ho-hum copying between data types.

By the way, I can feel you groaning at Listing 6.52. Developers *hate* writing “serialization” code like this. It’s tedious. It’s boring. It’s the worst of the mindless grunt work that’s so often involved in programming.

But it buys you freedom.

This minor tedium decouples us from the tyranny of what’s already there. It lets us stake a claim in the codebase and carve out a section we get to control and design. It starts small, but it grows over time. It’s how we make inroads into legacy codebases.

This translation from our semantic domain type back into the generic Invoice Entity lets us keep using all of the existing machinery in the codebase until we’ve matured enough to start using our own repositories.

It lets our save implementation to look like this:

Listing 6.53 Using existing machinery by mapping back to its world

```
private void save(LateFee<? extends Lifecycle> latefee) {
    invoiceRepo.save(
[CA]                toInvoice(latefee)); #A
    if (latefee.state() instanceof UnderReview review) {
        customerRepo.save(toCustomer(latefee)); #B
    }
}
```

#A Aside from the quick transformation from LateFee → Invoice, it looks like business as usual
#B (the toCustomer implemetation is omitted for brevity. It's just more of the same DTO mapping code)

6.6 Data is the ultimate interface

In a more real “real world” than the one we’ve painted in this chapter, we’d have a whole host of other concerns to handle that aren’t adequately addressed by our current single-threaded for-loop.

Listing 6.54 Potential failure points

```
public void processLatefees() {
    loadInvoicingData().forEach(data -> {
        LocalDate today = data.currentDate();
        EnrichedCustomer customer = data.customer();
        List<Invoice> invoices = data.invoices()
        List<PastDue> pastDue = collectPastDue(customer, today, invoices);
        LateFee<Draft> draft = buildDraft(today, customer, pastDue);
        ReviewedFee reviewed = assessDraft(data.rules(), draft);
        LateFee<? extends Lifecycle> latefee = switch (reviewed) {
            case Billable b -> this.submitBill(b); #A
            case NeedsApproval a -> this.startApproval(a); #A
            case NotBillable nb -> nb.latefee().markNotBilled(nb.reason()); #A
        };
        this.save(latefee); #A
    });
}
```

#A What happens if any of these fail?

What if one of the service calls fails? How do we handle retries? What about priority -- should Billing ever be blocked because the Approval service is down? Will all of this run fast enough? The list of questions goes on and on.

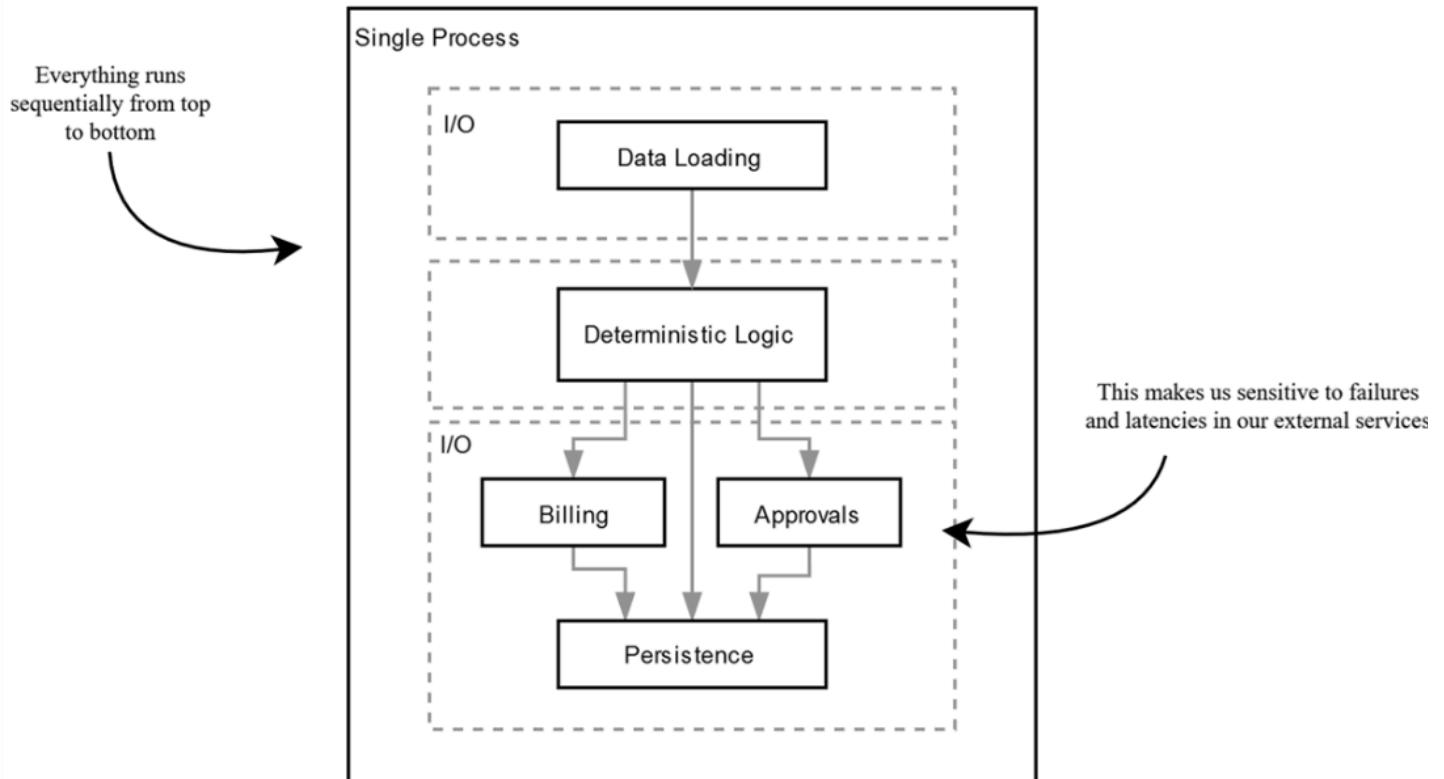


Figure 6.18 Doing everything in one big for-loop makes us sensitive to errors and service latencies

A hidden result of our modeling work is flexibility at the architectural level. Our data types have created “seams” throughout the application. We can snap off entire ideas in the system, like Billing, or starting Approvals, and throw them behind a queue, or run them parallel, or as their own micro-service. Further, because we’ve made the main logic deterministic, we have even more options about how it runs. Rather than a for loop, maybe we run in one big fat atomic transaction? Or batches? Or whatever makes sense.

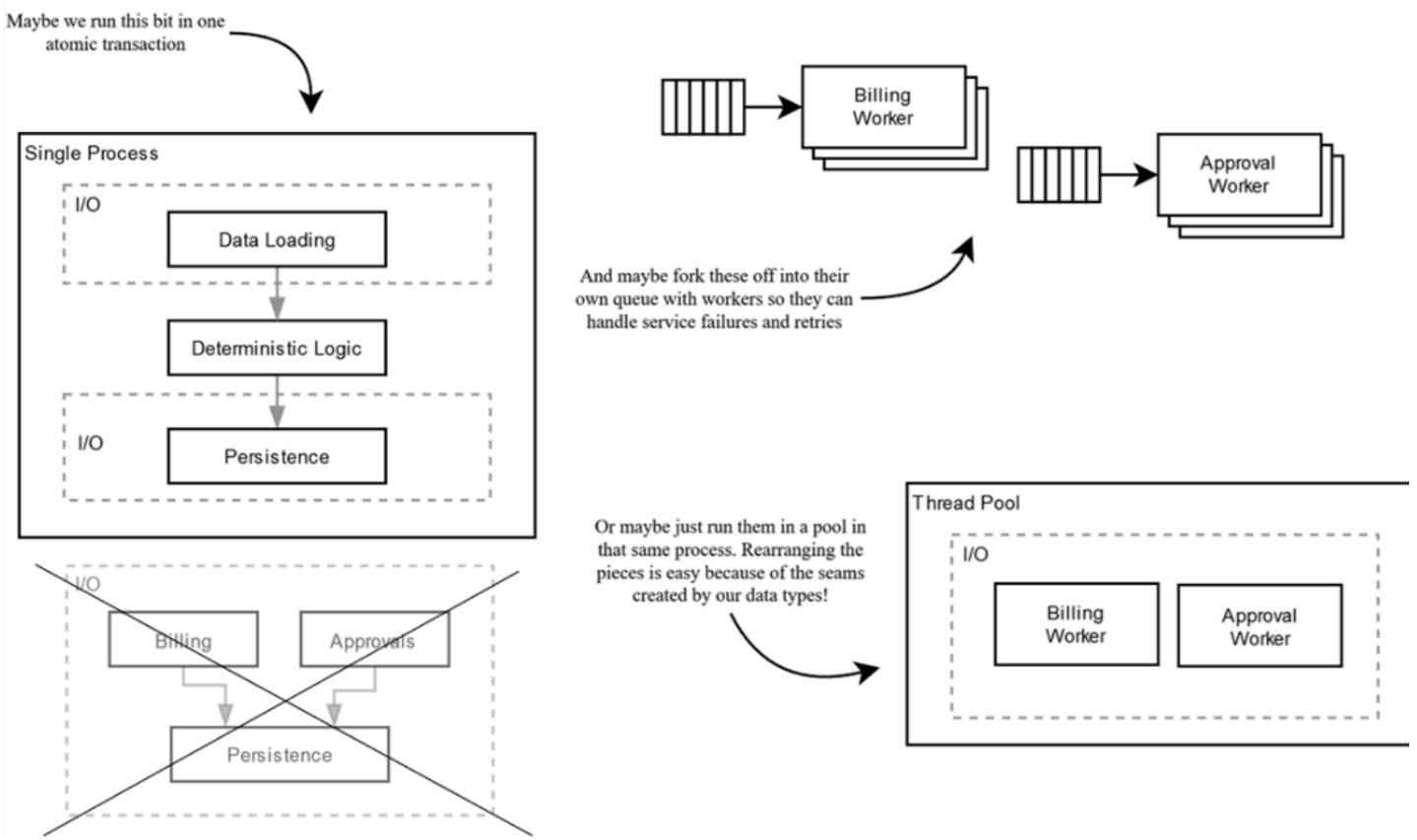


Figure 6.19 Rearranging the pieces is easy when everything speaks well-formed data

All of these become trivial refactorings because our data types have created natural interfaces in the code. A little bit of data goes a long way.

6.7 Putting it all together

Here's our finished Service logic.

Listing 6.55 Our feature in all of its semantic glory

```
static class LateFeeChargingService {  
    private ApprovalsAPI approvalsApi;  
    private BillingAPI billingApi;  
    private CustomerRepo customerRepo;  
    private InvoiceRepo invoiceRepo;  
    private CustomerStorageFacade customerFacade;  
    private RulesRepo rulesRepo;  
  
    public void processLatefees() {  
        loadInvoicingData().forEach(data -> {  
            LocalDate today = data.currentDate();  
            EnrichedCustomer customer = data.customer();  
            List<Invoice> invoices = data.invoices()  
            List<PastDue> pastDue = collectPastDue(customer, today, invoices);  
            LateFee<Draft> draft = buildDraft(today, customer, pastDue);  
            ReviewedFee reviewed = assessDraft(data.rules(), draft);  
            LateFee<? extends Lifecycle> latefee = switch (reviewed) {  
                case Billable b -> this.submitBill(b);  
                case NeedsApproval a -> this.startApproval(a);  
                case NotBillable nb -> nb.latefee().markNotBilled(nb.reason());  
            };  
            this.save(latefee);  
        });  
    }  
  
    public LateFee<? extends Lifecycle> submitBill(Billable billable) {  
        BillingResponse response = this.billingApi.submit(/*...*/);  
        return switch (response.status()) {  
            case ACCEPTED ->  
                billable.latefee().markBilled(new InvoiceId(response.invoiceId()));  
            case REJECTED ->  
                billable.latefee().markNotBilled(new Reason(response.error()));  
        };  
    }  
  
    public LateFee<UnderReview> startApproval(NeedsApproval needsApproval) {  
        Approval approval = this.approvalsApi.createApproval(/*...*/);  
        return needsApproval.latefee().markAsBeingReviewed(approval.id());  
    }  
  
    @Transaction  
    private void save(LateFee<? extends Lifecycle> latefee) {  
        invoiceRepo.save(toInvoice(latefee));  
        if (latefee.state() instanceof UnderReview) {  
            customerRepo.save(toCustomer(latefee));  
        }  
    }  
  
    Stream<InvoicingData> loadInvoicingData() {  
        LocalDate today = LocalDate.now();  
        Rules rules = rulesRepo.loadDefaults();  
        return customerFacade.findAll().map(customer ->  
            new InvoicingData(  
                today,  
                customer,  
                invoiceRepo.findInvoices(customer.id().value()),  
                rules  
        );  
    }  
}
```

```

    );
}

static Invoice toInvoice(LateFee<? extends Lifecycle> latefee) {
    Invoice invoice = new Invoice();
    invoice.setInvoiceId(switch (latefee.state()) {
        case Billed(var id) -> id.value();
        default -> Invoice.tempId();
    });
    invoice.setCustomerId(latefee.customer().id().value());
    invoice.setLineItems(List.of(new LineItem(
        null,
        "Late Fee",
        latefee.total().value(),
        Currency.getInstance("USD")
    )));
    invoice.setStatus(InvoiceStatus.OPEN);
    invoice.setInvoiceDate(latefee.invoiceDate());
    invoice.setDueDate(latefee.dueDate());
    invoice.setInvoiceType(InvoiceType.LATEFEE);
    invoice.setAuditInfo(new AuditInfo(
        null,
        latefee.includedInFee().stream().map(PastDue::invoice).toList(),
        switch(latefee.state()) {
            case Rejected(var why) -> why.value();
            case UnderReview(String approvalId) -> "...";
            default -> null;
        }
    ));
    return invoice;
}

Customer toCustomer(LateFee<UnderReview> latefee) {
    return new Customer(
        latefee.customer().id().value(),
        latefee.customer().address(),
        latefee.state().id()
    );
}
}
}

```

6.8 Wrapping up

This chapter walked through an implementation for the Late Fees data model we created in chapter 5. We covered a lot of ground, but at the heart of everything was the idea of determinism. This simple idea guided our implementation both in the small, where we used it to refactor and simplify our methods, as well as in the large, where we used it organize the layout of our codebase.

Next up, we're going to take a deeper dive into the “algebra” part of algebraic data types. We're going to learn that good engineers copy, but great engineers steal.

6.9 Summary

- Determinism is a powerful tool for simplifying an implementation

- “Function” has a semantic meaning independent of Java. It describes something that deterministically maps inputs to outputs.
- Everything in Java is technically a method behind the scenes (even things like `Function<A,B>`). We separate the semantics from the syntax when talking about functions.
- Making methods static is a useful convention for signaling that they will act as functions
- Functions have a finite “size” that’s dictated by their type signature
- We can limit possible implementations in a function by controlling its inputs and outputs
- The implementation for well-scoped functions tends to write itself. The types make it so that there’s only one possible implementation that will compile.
- Functions are behaviorally no different from a lookup table. We can freely swap between them with no effect on our program
- Functions are just data. We can take them as input and return them as output
- Functions model *relationships* between two pieces of data
- Use caution when trading Functions for Maps. It trades compile time guarantees for runtime assumptions
- Don’t get caught up in purity. “Basically deterministic” is close enough to “actually deterministic” for most programming.
- Organize your implementation around deterministic and non-deterministic functionality
- Non-Determinism makes everything harder. Refactor towards deterministic code.
- Types capture important invariants we know about the domain. This will create friction with the parts of your code that work on assumptions.
- Let uncomfortable disagreements over types drive your codebase in a better direction
- Not every disagreement is an emergency. Be measured. Often annotating and moving on is the best strategy
- Don’t get wrapped up in the Optional holy war. Focus on what your code communicates
- How we interact with Optional can control where we draw our reader’s attention
- It’s OK to introduce new types! As many as you need! Anywhere you need them!
- Code clarity trumps all other concerns
- The exchange points between our feature and the rest of the world will always be the trickiest. Give them plenty of thought.
- Treating the Entities outside of our feature as DTOs lets us reuse all of the existing ORM machinery until we’re ready to promote our own
- Designing around data builds natural seams into our application. It makes it trivial to refactor and rearrange the individual pieces

7 Guiding the design with properties

This chapter covers

- Formal definitions of correct
- Designing around algebraic properties
- Parallel universes and how to hop between them

Writing software is hard. It's an endless fight against complexity. We've spent years coming up with design patterns and frameworks to control this complexity, but the complexity is unyielding. Development slows. Bugs rear their heads. The gap between what the requirements say the software should do and what it actually does grows steadily over time.

We all want to keep the complexity at bay, but opinions differ on how to do it. How do you decide when a design approach is "better?" When it's SOLID? Or uses more OOP? Less? Design discussions often stalemate around mutual accusations of "complexity." One approach gets sanctified as "simple," all others denigrated as "complex."

But what does any of that mean? What is being measured?

In this chapter, we're going to tackle a meaning for "simple" that's based around algebraic properties. If we can reason about the set of values that our code denotes, and what the relationships between those values mean, it has earned the label of "simpler." If not, we keep digging until we find it.

Writing clear code requires clear thinking. Algebraic reasoning is how we do it.

7.1 We begin with an exercise

We're going to do some ETL in the same finance domain from the previous chapters. We need to load customer data from one place, transform it, and stick it somewhere else. But we have a problem: the data is garbage. The upstream is filled with conflicting records.

So, we'll find data like this:

The data is in bad shape. We find multiple rows for the same primary key

ID	Audit Finding	Policy	Premium
1	OUT_OF_COMPLIANCE	FLEXIBLE	Y
1	INACCURATE	FLEXIBLE	N
1	NO_ISSUE	FLEXIBLE	Y
2	NO_ISSUE	IMMEDIATE	N

Figure 7.1 A first look at the world we're in

Multiple rows have the same ID and each holds different data. There's no clear way of picking the "right" one. But at least we *have* data. Sometimes we'll find records with nothing in them like this:

More duplicated keys that conflict with each other

ID	Audit Finding	Policy	Premium
7	NO_ISSUE		
7			N
7		FLEXIBLE	
8	NO_ISSUE	IMMEDIATE	N

And sometimes those conflicting records will be filled with nulls

Figure 7.2 A sea of missing and undefined data

Is a row with a value more correct than a row without one? Do nulls represent *removals* of values or missing values? There's no way to tell. The data is junk.

So, the business people come up with a simple rule: pick the option that favors *not* billing the customer. We don't know what bedrock truth is, but we can construct a version of world that keeps the customer happy.

Table 7.1 A few requirements

A0	Remove all conflicting duplicate records
A1	Replace nulls with concrete values
A2	If two values conflict, pick the one that favors not billing the customer
A3	If two values have the same customer impact, resolve via sort

It sounds simple, but these requirements can be deceptive. It's surprisingly easy to trip into complex implementations if you're not careful.

To show this, we're going to start off with a small exercise. I'm going to give you a starting point. Listing 7.1 has all the data models, plus two functions that will tell you whether a policy or audit finding is considered as favoring or harming the customer. It has everything you'd need in order to satisfy the requirements.

Listing 7.1 A starting point

```
public enum Policy {          #A
    GRACE_PERIOD,
    FLEXIBLE,
    IMMEDIATE,
    STRICT,
    MANUAL_REVIEW
}

public enum AuditFinding { #A
    BILLING_ERROR,
    OUT_OF_COMPLIANCE,
    INACCURATE,
    NO_ISSUE
}

record RawData(           #B
    String id,
    Optional<Policy> policy,
    Optional<AuditFinding> findings,
    Optional<Boolean> isPremium      #C
) {}

enum CustomerImpact { HARMS, FAVORS };                      #D
static CustomerImpact policyImpact(Policy stage) {          #D
    return switch (stage) {
        case GRACE_PERIOD, FLEXIBLE -> CustomerImpact.FAVORS;
        default -> CustomerImpact.HARMS;
    };
}
static CustomerImpact findingsImpact(AuditFinding stage) { #D
    return switch (stage) {
        case NO_ISSUE, INACCURATE -> CustomerImpact.FAVORS;
        default -> CustomerImpact.HARMS;
    };
}

static List<RawData> cleanDuplicates(List<RawData> rows) { #E
    // [your implementation here]
}
```

#A These enumerations model the potential values in each column.

#B This is the core model for representing an individual row in our data set

#C Note that the Boolean is also wrapped in an optional. “We don’t know” is a valid state

#D These functions will tell you whether a given Policy or AuditFinding “favors” or “harms” the customer.

#E Here’s an example starting point. We’re ignoring how we get this data so we can focus just on what we do with it. So... what do we do with it?

Here’s the exercise: think about how you would implement this. And notice your thinking as you do.

Where does your mind go?

Most of us will start visualizing code. I think about the list of data, then bucketing it to find the duplicates, then maybe seeing if I can get rid of any nulls, and so on. You might start somewhere else, but I bet that it similarly starts with code.

Here’s another question: how would you test it?

This can be harder to visualize. Most of the time our tests end up tied to our specific implementation. We might be able to pull out a few high-level business test cases from the requirements, but the bulk of the tests we write end up about reaching "coverage." They tell us a specific line in our code does something, but they don't tell us if that something is *correct*.

That leads to the last question in the exercise: What does it mean for an implementation to be *correct*?

Correctness can be hard to define if you've never tried before. It's easy to get wrapped up in strange cyclic definitions where the code is its own definition of correct. "The implementation is correct because the tests we wrote against the implementation say that it does what it does." Or we get stuck with definitions that don't say anything at all: "The code is correct when it satisfies the requirements." Ok. But what does *that* mean? How would you use that definition to make sure your implementation is correct?

These questions are all reflections of each other if we think *algebraically*. Answering one answers the others. Exploring one informs the others. Like most of the ideas in this book, it requires taking a step away from the details of the code and instead focusing on what the code should *mean*. To think algebraically is to think about the equations and laws that govern our data. If we know those laws, then we can use them to write tests that tell us if our code is correct.

To do this we have to use some math. Nothing hard. Some logic and a dash of set theory. Everything we need you already know because you write Java. You use algebraic ideas every day, they're just hidden away in the details of the language. We're going to bring them to the surface. Doing so allows us to be rigorous about what we want the code to *do*. Once we know that, the rest is easy.

This is way less sophisticated than it might sound. Despite what words like "Algebra" might cue, this chapter holds no esoteric abstractions or formulas (which is great, because I wouldn't understand them). You can think of them as really basic design patterns. Despite being from math, they're found everywhere in Java. Learning to tap into them opens up one of software engineering's most prized possessions: reuse. Most of what we need has already been written. It's sitting inside of Java's standard library waiting for us. Algebra is how we unlock it.

7.2 Thinking Algebraically

The basic definition of an "algebra" is that you have some set of things (Numbers, Strings, Images), and some operations that we can apply to those things (add, concat, flip), and some rules that those operations must follow (we'll get to those in a second).

This should feel familiar. Object oriented design works the same way. First, we pick out the nouns (sets), then ask what they can do (verbs / operations). The two approaches differ only in the constraints. Algebras are built on deterministic *functions*. Data in; data out. No side-effects. No void methods.

This constraint allows us to reason about code *equationally*. Equations let us notice that two pieces of code, no matter how different they might appear, are functionally the

same. And the result of this noticing is, as we'll see, the ability to substitute complex expressions for simpler ones.

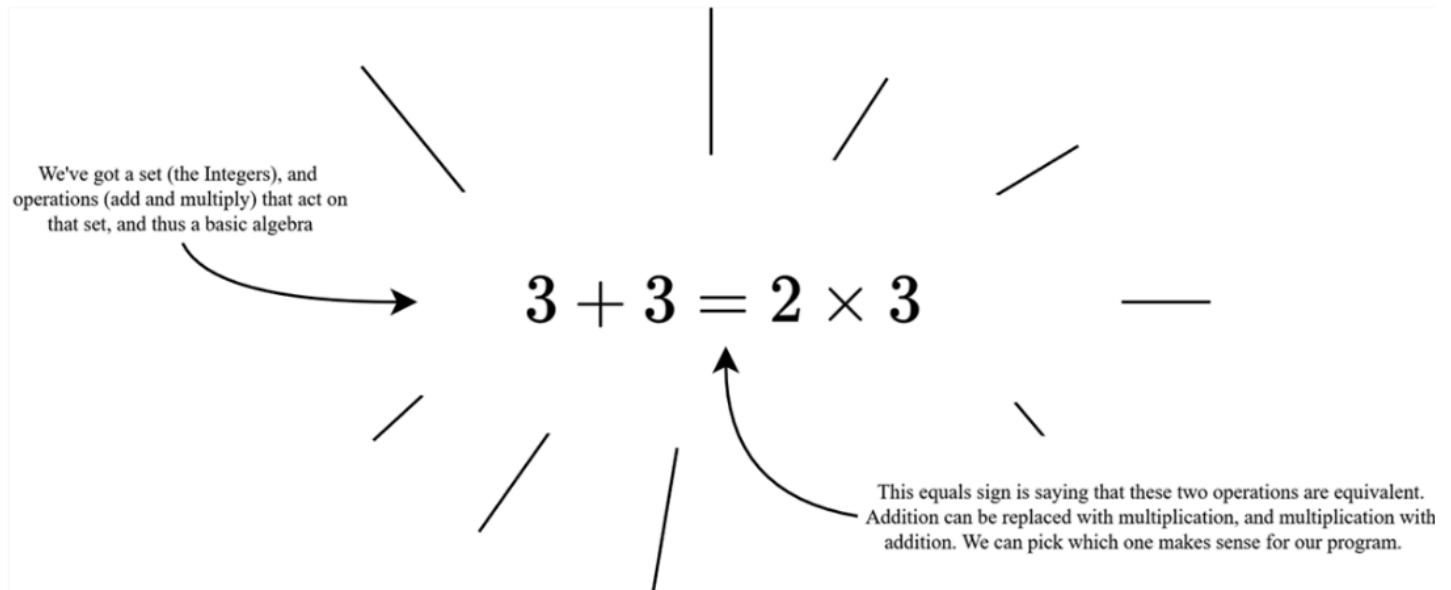


Figure 7.3 An exciting first example!

Equational reasoning enables us to hunt for laws that describe what our operations mean. Not for a specific input, but *for all possible* inputs. Algebra leads to universal abstractions.

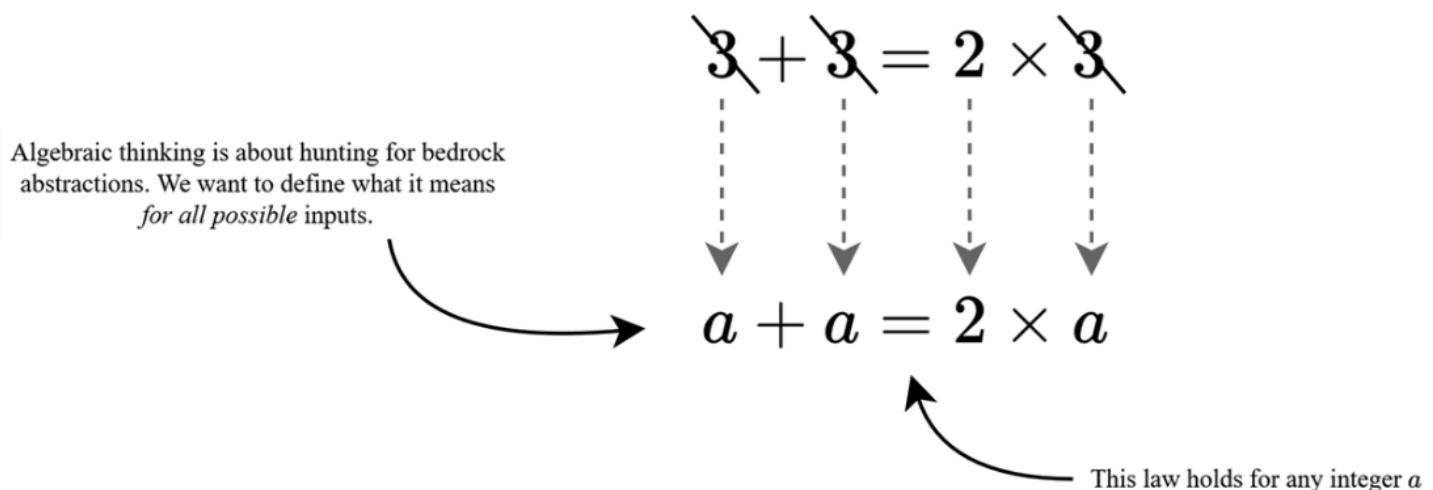


Figure 7.4 Algebraic laws must hold for all possible inputs

To "do" algebra in a programming context, all we have to do is setup an equation between two pieces of data and then ask our favorite question: "what does it mean?"

In our current domain of merging conflicting duplicated rows, we would ask: What would it mean to "add" two conflicting records?

We've got a set (the duplicated data) and an operation (adding two rows) and thus the start of a basic algebra

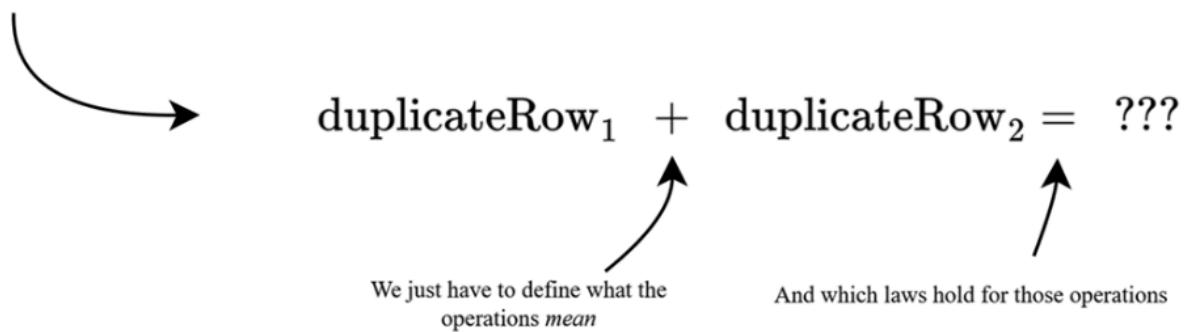


Figure 7.5 An algebra appears!

And now we're officially thinking about our problem algebraically.

We haven't actually done anything yet, but there are a few cool things to notice.

The first is how unsophisticated it is. We slapped a $+$ symbol between two records. That's it. Now we're "doing" Algebra. If you were worried about the mention of "math" at the beginning of the chapter, fear no more. This is as complicated as it will get.

The second is how *small* it is. We're not thinking about lists, data structures, or telling the computer "how." Framing our problem algebraically focuses us on the most important part: what it means to combine two data types.

Lastly, notice that this algebra is giving us a "language" in which we can talk about correctness without talking about implementation. We're working above the code in a world where we can think and explore what *should* hold for any implementation – not just the one we happen to write. It's loose right now, but we're going to learn how to make this air tight.

Where do we go from here? How do we define what this plus symbol should mean? What should it return? What are the laws?

Turns out, we can just steal.

7.2.1 Standing on Shoulders, Not Toes

Mathematicians stand on each other's shoulders and computer scientists stand on each other's toes.

Richard Hamming

Programmers have had about 80 years since the dawn of computing to come up with design patterns. Mathematicians have had thousands of years. Designing around an algebra gives us access to their work. There's a whole world of shoulders to stand on.

One of the simplest and most useful "patterns" we can explore is the humble binary operation. It's the meat and potatoes of the algebra world and powers just about everything.

Here's a binary operation performing addition on the integers.

The addition operator is *binary*. It takes two values from the same set as "input"

And gives back a value in this same set as "output"

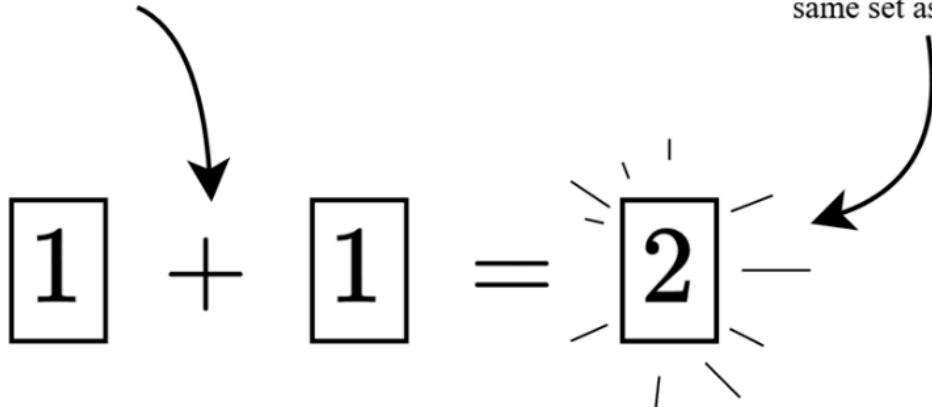


Figure 7.6 Addition is a closed binary operation

Binary operations take two values, both of the same type, and return a new value, also of the same type. They're built into Java as a specialization of the `BiFunction` interface. It looks like this:

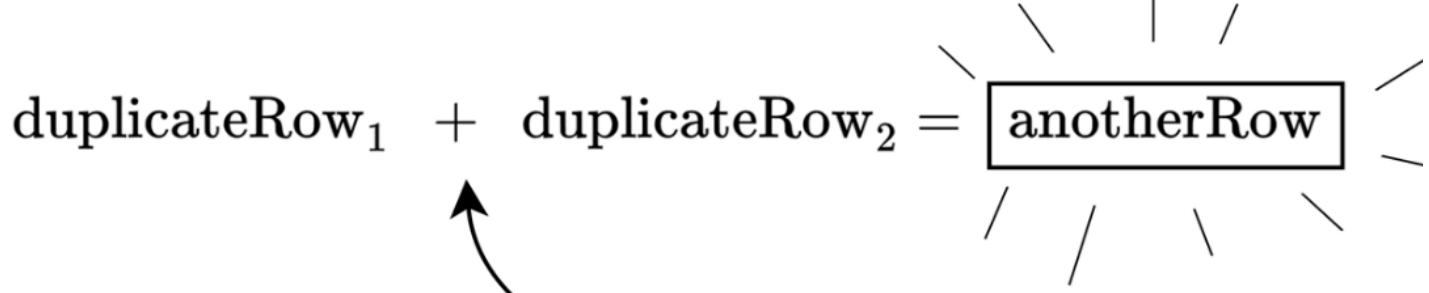
Listing 7.2 Java's BinaryOperator interface

```
@FunctionalInterface  
public interface BinaryOperator<T> extends BiFunction<T,T,T> {} #A
```

#A Binary operations take and return the same type

We can take this "design pattern," simple as it is, and use it as the core around which we design our feature.

We can steal ideas from other algebras and design our "algebra of merging duplicates" as a closed binary operation over our Row data type



We don't know what this operator does yet, but we do know that it behaves correctly when it makes a new row

Figure 7.7 Designing our algebra by copying existing, well-trodden design patterns

In Java, it would look like this.

Listing 7.3 Translating into Java

```
public static RawData add(RawData x, RawData y) {...} #A
```

#A Designing our “algebra of merging duplicates” as a closed binary operation

It’s not earth shattering, but this small choice connects us to a world of extraordinary power. Learning to notice these binary operations is an important skill to develop. They show up absolutely everywhere in Java. Many are familiar and obvious.

Listing 7.4 A few familiar binary operations

```
assertEquals(1 + 1, 2);
assertEquals(2.12 + 5.3, 7.42);
assertEquals("Hello" + "World", "HelloWorld");
```

Others can be less obvious because of their naming or operators.

Listing 7.5 More Binary operations

```
Stream.concat(Stream.of(1,2), Stream.of(3)) #A
true || false #A
true && false #A
```

#A Despite the differing names and symbols, these are all binary operations!

Some operations will require us to mentally rearrange the code in order to see them. Java is built around objects, so we often end up with something that’s spiritually a binary operation exposed as a unary (single argument) operation off of an existing object.

Listing 7.6 Binary Operations exposed as Unary Operations off objects

```
BigDecimal.ONE.add(BigDecimal.ONE) #A
Function<Integer, Integer> addOne = (x) -> x + 1;
Function<Integer, Double> divideBy10 = (x) -> x / 10.0;
addOne.andThen(divideBy10) #B
```

**#A Sometimes you’ll have to squint to see the binary operations. Even though this only accepts a single argument (a Unary Operator), the other argument is being provided implicitly from the object itself.
#B Same thing here. Despite how we interact with it in Java, this completely honors the binary operation contract.**

Finally, most operations won’t be in the code at all. We have to see “through” an implementation into the algebraic properties that sit behind it. If you can notice that a chunk of code fits into an algebraic shape, you immediately gain access to all of the laws and tools for reasoning that thousands of years of mathematical refinement have wrought.

7.3 Noticing algebraic operations in every day code

Everyday code is filled with algebraic operations. They just take some squinting to see.

To build up our noticing skills, let’s explore this code from an algebraic perspective.

Listing 7.7 Control logic for cleaning the duplicate records

```
static List<RawData> cleanDuplicates(List<RawData> rows) { #A
    Map<String, List<RawData>> dupesById = new HashMap<>(); #B
    for (RawData row : rows) {
        String key = row.id();
        if (!dupesById.containsKey(key)) {
            dupesById.put(key, new ArrayList<>());
        }
        dupesById.get(key).add(row);
    }

    List<RawData> cleaned = new ArrayList<>(); #C
    for (List<RawData> duplicates : dupesById.values()) { #D
        RawData merged = duplicates.getFirst();
        for (RawData other : duplicates.subList(1, duplicates.size())) { #E
            merged = add(merged, other); #E
        }
        cleaned.add(merged); #F
    }
    return cleaned;
}
```

#A The main control logic for our feature

#B We start by grouping all of the data by ID to find the duplicates

#C Setting up our output data set

#D Now we loop over the list of values stored under each key

#E And then loop over each sublist and apply our binary operation to reach row

#F Then finally add the now combined result to the output data

If you're familiar with Java's Collectors, you'll probably want to start replacing these loops with streams. `groupingBy` could find the duplicates. `map` and `reduce` could handle everything else. However, we're going to hold off for now; doing such refactors too soon would rob of us what we want to accomplish – noticing the algebra hidden in everyday code.

This code, despite its very imperative appearance, is filled with potential algebraic operations. Learning to identify them is the initial challenge. Most code in Java is written like Listing 7.7. It's preoccupied with managing what the computer is doing and what it's doing it to. It's not bad code by any means, but it shifts "where" we reason. Our thinking gets pulled down into the world of side-effects and state updates. And once you're down in that level, your tests become about that level, too. You get coupled to the details of how things get done, rather than what they *should* be doing.

Thus, the importance of building up muscles that let us see "through" a piece of imperative code into the underlying algebra that powers it. It allows us to reason at a higher level of abstraction, even if the code itself remains interested in shuffling bits (which is fine!).

Here's the basic strategy for building up your noticing skills: hunt for the binary operations. Often, code will have something *almost* binary operation-like, that, with a small tweak, and a bit of imagination, can be refactored into a real binary operation.

And once you notice one, you tend to start noticing others.

To see what I mean, let's extract the chunk of logic that groups the duplicates into its own function.

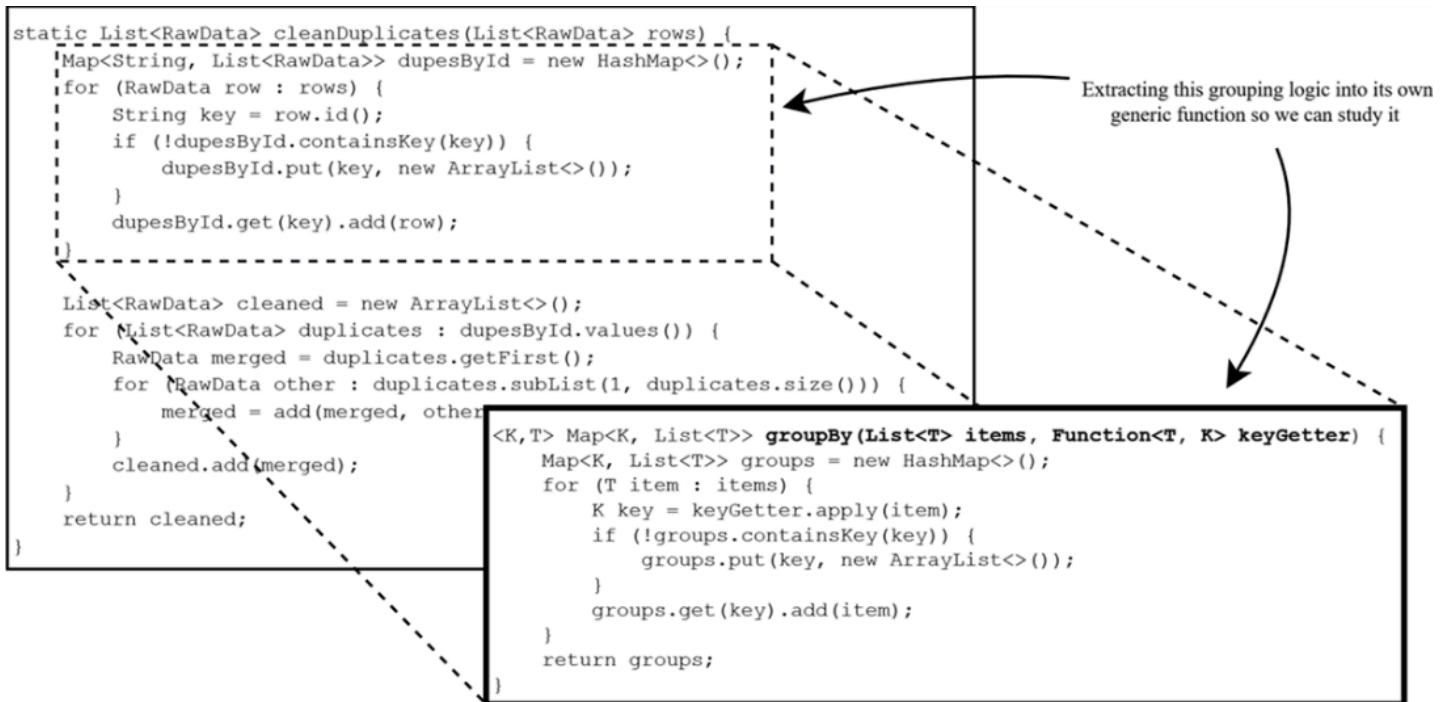


Figure 7.8 Extracting the grouping logic into its own generic function

Specifically, let's look at how it collects items into lists. Right now, it's very focused on managing state. It seeds the map with an empty list, then grabs its reference, mutates it to add the new item, and then loops back around and do it all over again.

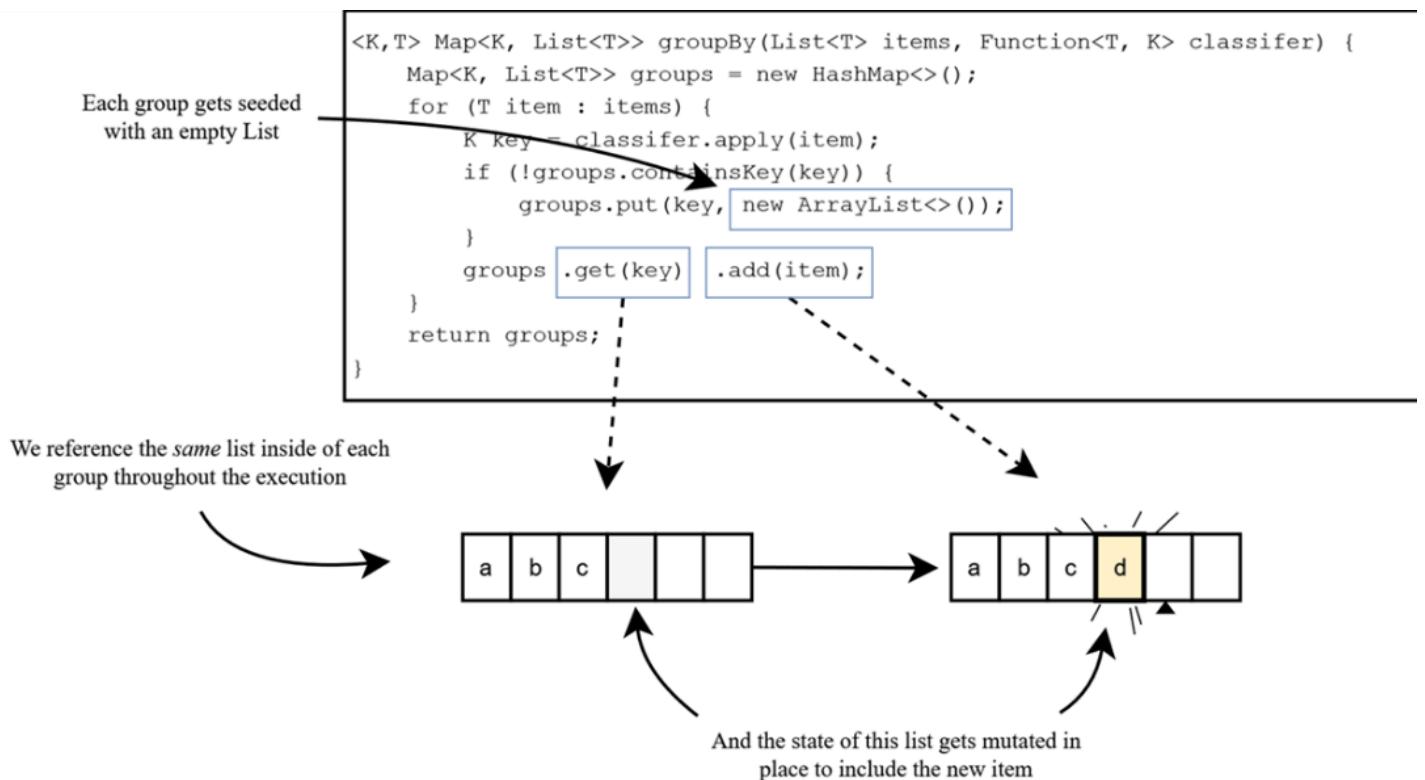


Figure 7.9 the state management involved in grouping

But if you do a bit of mental rearranging, you can squeeze all this side-effecting stuff into a vaguely binary operation(ish) shape.

With a bit of mental rearranging, we can visualize every-day methods as algebraic binary operations

```
<K, T> Map<K, List<T>> groupBy(List<T> items, Function<T, K> keyGetter) {
    Map<K, List<T>> groups = new HashMap<>() ;
    for (T item : items) {
        K key = keyGetter.apply(item);
        if (!groups.containsKey(key)) {
            groups.put(key, new ArrayList<>());
        }
        groups.get(key).add(item);
    }
    return groups;
}
```

Once it's in an algebraic "shape," we can see what we'd need to change in order to actually make it into a valid algebraic binary operation.

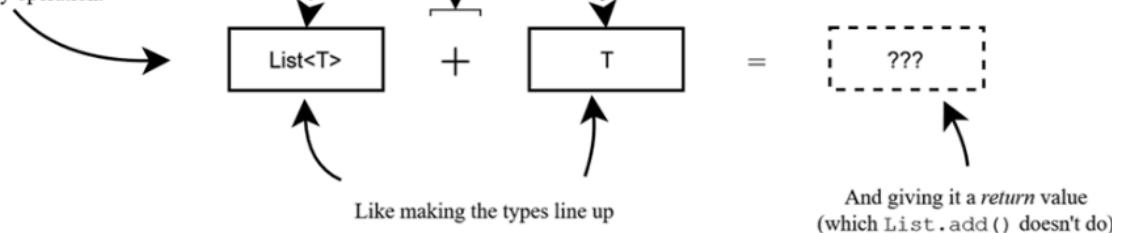


Figure 7.10 Visualizing common methods as algebraic binary operations

And once you get it into the rough shape, it's usually clear on how to transform it into a proper binary operation. For instance, the "problem" with `List.add()` from an algebraic perspective is that it's not *closed*. Its arguments don't line up and it doesn't return anything. But if we fix those, we get this interesting algebraic operation that combines lists.

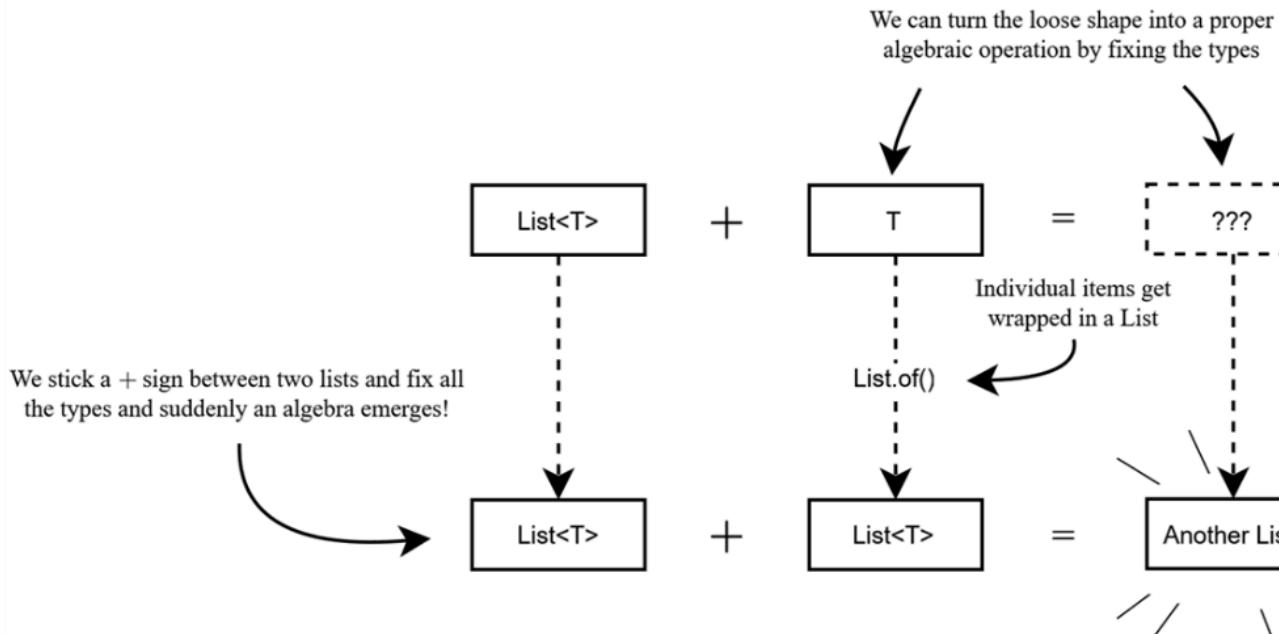


Figure 7.11 translating into a binary operation

We could write this "addition" operation something like this.

Listing 7.8 Adding two lists as a binary operation

```
static <T> List<T> add(List<T> list1, List<T> list2) { #A
    return Stream.concat(list1.stream(), list2.stream()).toList(); #A
}

static <T> List<T> add(List<T> list1, List<T> list2) { #B
    List<T> output = new ArrayList(list1);
    output.addAll(list2); #C
    return List.copyOf(output);
}
```

#A We might implement it like this

#B or like this -- or some other way. There are no rules here as long as it's a deterministic function

#C Does it matter that we use mutation internally while writing a deterministic function that takes and returns immutable data? I'd argue no! Don't let purity get in the way of practicality.

And refactor to use this in our implementation

Listing 7.9 Refactoring groupBy to combine lists with algebraic Binary Operations

```
<K,T> Map<K, List<T>> groupBy(List<T> items, Function<T, K> classifier) {
    Map<K, List<T>> groups = new HashMap<>();
    for (T item : items) {
        K key = classifier.apply(item);
        List<T> list1 = groups.getOrDefault(key, new ArrayList<>()); #A
        List<T> list2 = List.of(item); #B
        groups.put(key, add(list1, list2)); #C
    }
    return groups;
}
```

#A Rather than putting an empty list into the map and mutating it in place, we read the value out as data

#B Our "algebra" is about Lists, so we wrap the incoming item in a List

#C We combine the two lists together and store the new list in the map.

On its own, this refactor isn't terribly interesting. Actually, if we judge it in isolation, we've made the code slightly worse. It's less efficient. It's less clear. It's kind of weird looking. But it's neat that we *could* make this transformation with just a bit of massaging. There was a small algebra hidden inside of that imperative implementation. Noticing it allowed us to take something that mutates state and replace it with something that computes data.

And once you notice one, you can start noticing more.

Map is doing the same *almost* binary operation-like thing as List. It's just harder to see because the `put()` API doesn't really fit the algebraic shape. However, if we swap to its cousin, `putAll()`, something very binary operation-like emerges. We can use that as a jumping off point.

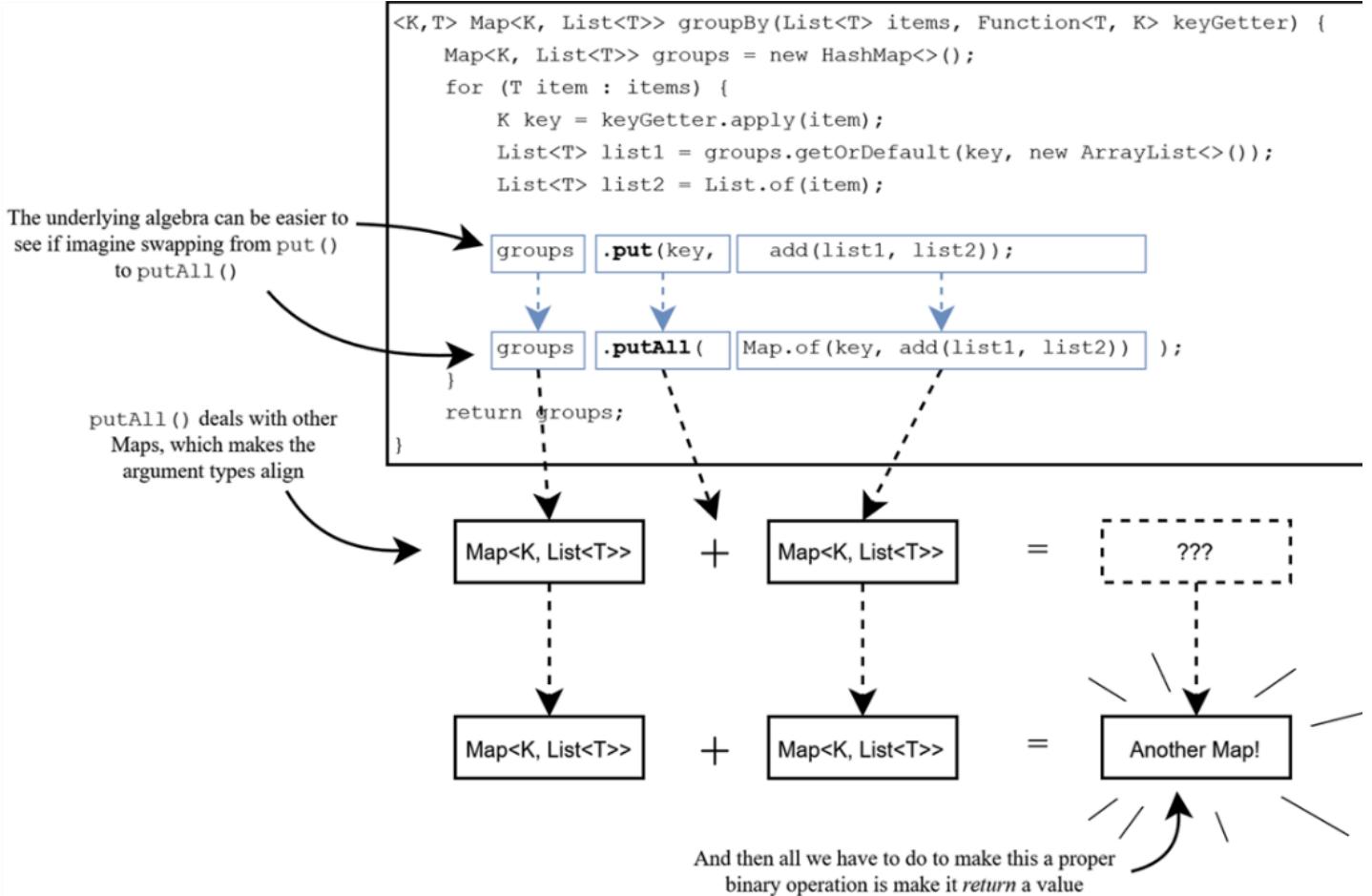


Figure 7.12 Visualizing adding something to a map as a binary operation combining maps

There's an interesting pattern worth noticing here. Rather than mutating some object in place, we can accomplish the same result "algebraically" by expressing the change we want as an "addition" operation against immutable data.

There are a few ways we could implement "addition" on maps because we have to decide how to handle collisions. The first option is to do nothing; discard the incoming value. Or we could go the other way: replace one value with the other.

Listing 7.10 A few of the ways we could implement Map “addition”

```
static <K,V> Map<K,List<V>> add( #A
    Map<K, List<V>> left,
    Map<K, List<V>> right) {
    HashMap<K, List<V>> combined = new HashMap<>(right);
    combined.putAll(left) #B
    return combined
}

static <K,V> Map<K,List<V>> add( #C
    Map<K, List<V>> left,
    Map<K, List<V>> right) {
    HashMap<K, List<V>> combined = new HashMap<>(left); #D
    combined.putAll(right)
    return combined
}
```

#A This is an example of a “left biased” implementation

#B It overwrites any conflicting values in “right” with values from “left”

#C Here’s a “right biased” flavor

#D It overwrites any values in the left map with values from right

These implementations are useful, but what if we don’t want to throw away any data? We basically have one option: combine them with a binary operation.

Listing 7.11 Combining Maps and their values with binary operations

```
static <K,V> Map<K,List<V>> add(
    Map<K, List<V>> left,
    Map<K, List<V>> right,
    BinaryOperator<V> operator #A
) {
    HashMap<K, List<V>> combined = new HashMap<>(left);
    right.forEach((key, value) -> combined.merge(
        key,
        value,
        (list1, list2) -> operator.apply(list1, list2))) #B
    return combined;
}
```

#A Supplying a binary operation as an argument

#B Now the code can decide what to do whenever collisions are encountered

This minor change turns out to be powerful. To see why, let’s refactor our `groupBy` implementation to use this new operation – to express map addition algebraically.

Listing 7.12 Expressing groupBy in terms of binary operations on Maps

```
static <K,T> Map<K, List<T>> groupBy(
    List<T> items,
    Function<T, K> classifier
) {
    Map<K, List<T>> result = new HashMap<>();
    for (T item : items) {
        K key = keyGetter.apply(item);
        Map<K, List<T>> anotherMap = Map.of(key, List.of(item)); #A
        result = add(      #B
            result,
            anotherMap,
            (list1,list2) -> add(list1, list2)); #C
    }
    return result;
}
```

#A First we create a new map with the updated data

#B Then produce another map that's the combination of what we have so far plus the new value

#C We can combine the lists

This refactor might seem like it's heading in the wrong direction, the code is looking less and less like normal Java, but something pretty remarkable is going on. We've described a sophisticated idea like `groupBy` with nothing more than a few binary operations. Our entire API is the `+` operator. We traded manual state management for a higher level of abstraction that describes what we want to do (combine data) rather than how we'll do it (mutating lists and Maps).

Now that we've refactored to expose the underlying binary operations, let's bring back the stream tooling. Converting our for-looping to map and reduce will make our next round of noticing easier.

Listing 7.13 Replacing manual looping with map and reduce

```
static <K,T> Map<K, List<T>> groupBy(
    List<T> items,
    Function<T, K> classifier
) {
    items.stream()
        .map(item -> Map.of(classifier.apply(item), List.of(item))) #A
        .reduce(Map.of(), (a,b) -> add(a, b, add)) #B
}
```

#A First we turn everything into a Map

#B Then we loop over it to combine it with our binary operations

The stream APIs make the pattern easier to see. First, we transform the data into a shape our algebra understands, then we do work by repeatedly applying binary operations on it.

This pattern shows up everywhere once you start seeing things algebraically. It starts small with things like Lists and Maps, but once you notice them there, you'll start noticing them elsewhere – including entire features.

Using binary operations to guide a design

Can we describe our *entire* problem just in terms of just adding two data types together?

Let's look at the main control logic one more time.

Listing 7.14 The same as what we started with, but using our custom groupBy

```
static List<RawData> cleanDuplicates(List<RawData> rows) {
    Map<String, List<RawData>> dupesById = groupBy(rows, rawData::id); #A

    List<RawData> cleaned = new ArrayList<>();
    for (List<RawData> duplicates : rowsById.values()) {
        RawData merged = duplicates.getFirst()
        for (RawData other : duplicates.subList(1, duplicates.size())) {
            merged = add(merged, other);
        }
        cleaned.add(merged);
    }
    return cleaned;
}
```

#A Using the groupBy method we defined above.

It's a fairly involved pipeline of transformations. We group, we loop, we loop some more, we combine, and then collect.

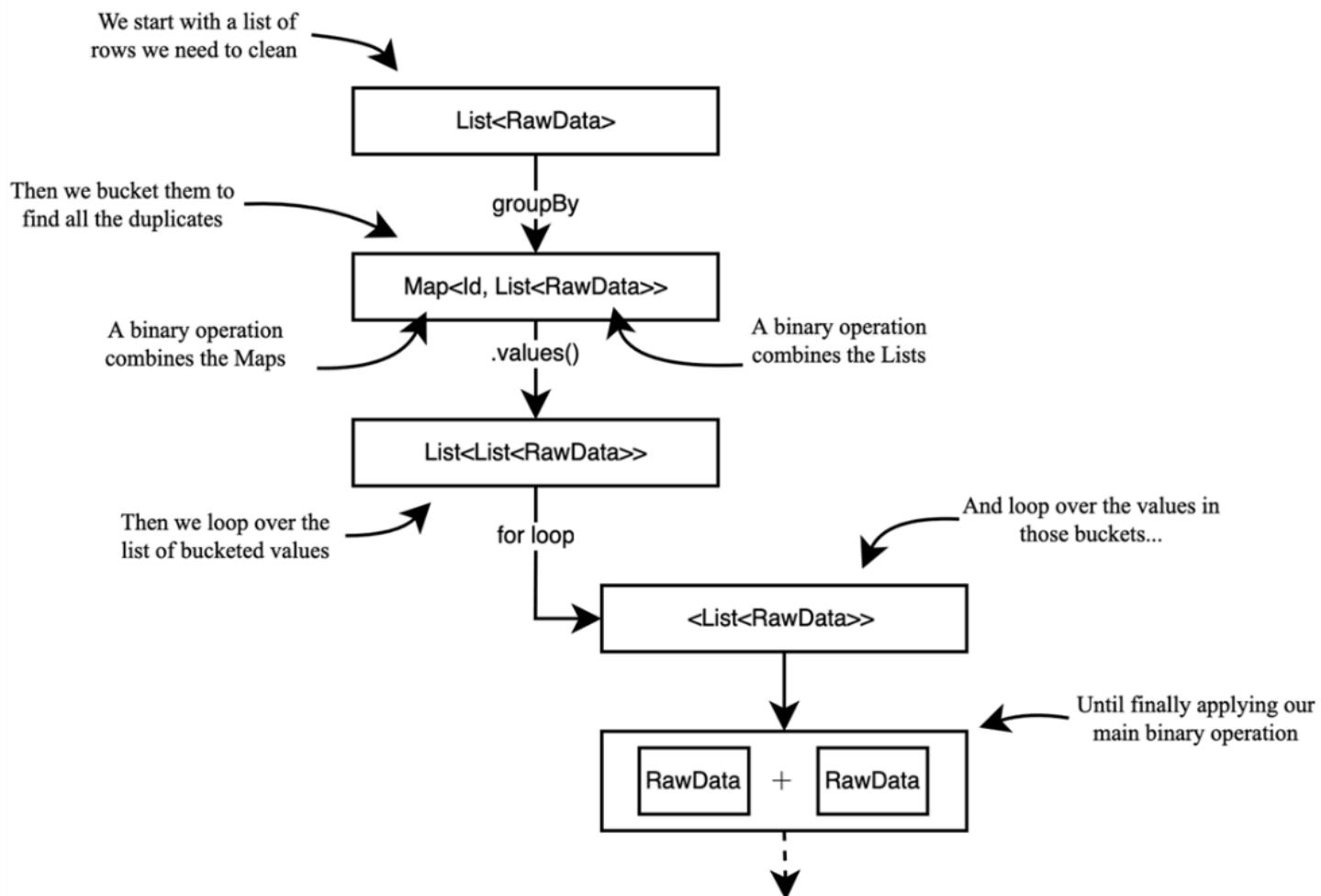


Figure 7.13 How we're pipelining our work

Even if we refactor everything to use the streams APIs, we still end up with the exact same set of transformations, just wrapped in a better pair of clothes.

Listing 7.15 Refactoring to use the streams APIs

```
record NonEmptyList<A>(List<A> items) {      #A
    NonEmptyList {
        if (items.isEmpty()) {
            throw new IllegalArgumentException(...);
        }
    }
}

static RawData merge(NonEmptyList<RawData> dupes) {      #B
    RawData initial = dupes.values().getFirst();
    return dupes.stream().skip(1).reduce(initial, Chapter7::add);
}

static List<RawData> cleanDuplicates(List<RawData> rows) {
    Map<String, List<RawData>> dupesById = rows.stream()  #C
        .collect(groupingBy(RawData::id));

    return dupesById.values().stream()      #D
        .map(NonEmptyList::new)
        .map(dupes -> merge(dupes))
        .toList();
}
```

#A A helper data type to capture that a List is guaranteed to have something in it.

#B We pulled the main combining work into its own function

#C Grouping using the built in groupingBy rather than our version

#D We grab the duplicates then map over them to reduce down to a final answer

Each action seems like it needs to be there. We can't "just add" two records, because not every record will have the same ID, so you "have" to group them first. And now you've got nested lists, so you've got to handle those, and so on for each step in the pipeline.

But this brings us to the point of all our "noticing" effort. If you take the algebraic view of the world, what do those operations actually *do*? `groupBy` just loops over the data and applies a binary operation. `reduce?` It also just loops over the data and applies a binary operation. We're going through an entire series of transformations just to end up back where we started: applying binary operations. So, what the heck are we doing?

Do we need to do all of this work? Is there a way we can express our problem as just a single binary operation?

It turns out we already stumbled on a way.

Check this out:

Listing 7.16 We loop over the items and apply a binary operation

```
static <K,T> Map<K, List<T>> groupBy( #A
    List<T> items,
    Function<T, K> classifier
) {
    return items.stream()
        .map(x -> Map.of(classifier.apply(x), List.of(x))) #B
        .reduce(Map.of(), (map1, map2) -> add( #C
            map1, map2
            Chapter7::add));
}
```

#A Inside of groupBy we loop over the data and apply binary operations

#B Right now, those binary operations are “about” combining lists, but they don’t have to be!

#C Everything else is just looping over the elements and apply a binary operation. It doesn’t care what that operation does

Inside of `groupBy` is this wonderfully general pattern that puts data into a certain shape, and then applies binary operations on it. Despite its simplicity, it can do almost anything. The current implementation is only “about” lists right now because we wrap the incoming items. But it doesn’t have to be.

We can make a new function that has this same shape, but that lets us choose which binary operation gets applied while constructing the map.

Listing 7.17 We loop over the items and... apply a binary operation!

```
static <K,V> Map<K,V> toMap(
    List<V> items,
    Function<V, K> classifier,
    BinaryOperator<V> binop #A
) {
    return items.stream()
        .map(item -> Map.of(classifier.apply(item), item)) #B
        .reduce(Map.of(), (m1, m2) -> add(m1, m2, binop)); #C
}
```

#A Supplying a binary operation that we can use while building the Map

#B No more wrapping the incoming item in List!

#C Conflicts are resolved via binary operation

And that leads to something remarkable.

If we turn our data into a map, and we have a way of combining those maps, and a way of combining the `RawData` inside of those maps, then there’s nothing else to do. That *is* the logic. Our entire feature, from the detection of duplicates, to their grouping, to their eventual merging is all handled automatically just by the algebraic act of “adding” Maps together.

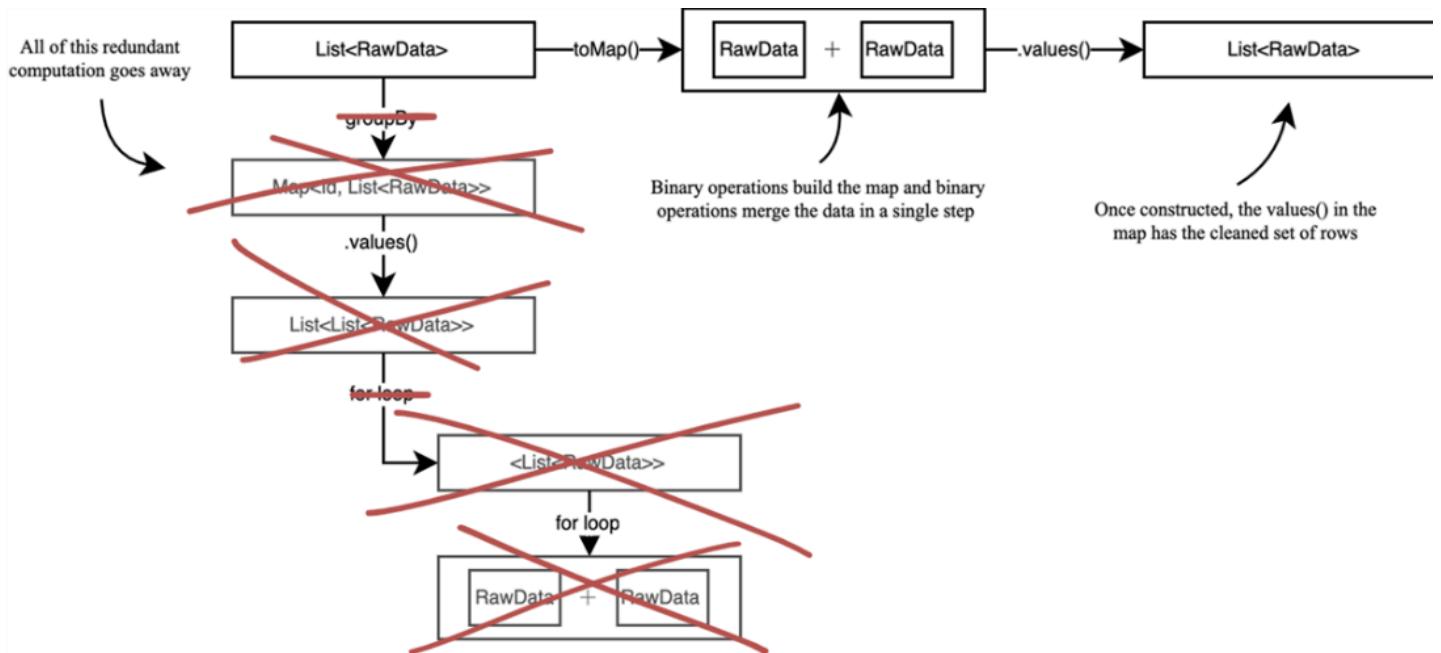


Figure 7.14 Algebraic reasoning leads to deep insights that can remove redundant computation

Once you notice the pattern, the entire implementation collapses down to a single line like this:

Listing 7.18 We loop over the items and, that's right, apply a binary operation!

```
static List<RawData> cleanDuplicates(Collection<RawData> rows) {
    return List.of(toMap(rows, (rowA, rowB) -> add(rowA, rowB).values()));
}
```

How cool is that? It accomplishes so much with so little. You might even call it “elegant.” Algebraic approaches can reveal *universal patterns* in your program. Things which can appear different from an algorithmic standpoint can suddenly appear less so an algebraic one. Noticing them can feel like stumbling upon a discovery. And once you begin to notice, code often goes through a rapid collapse towards simplification.

This code is beautiful, but we still haven’t answered one of the core things we’ve set out to answer: is this implementation correct? To answer that, we have to dig into the properties that make these operations work.

7.4 The algebraic properties of binary operations

Binary operations are more subtle than they first appear. Despite being made up of just two arguments, we can combine those arguments in all kinds of ways. It starts off simple enough; a single invocation of a binary operation has just two options.

Listing 7.19 Calling a binary operation with the same arguments in different orders

```
RawData row1 = new RawData(...);
RawData row2 = new RawData(...);
add(row1, row2); #A
add(row2, row1); #B

#A row1 then row2
#B row2 then row1
```

But once you start chaining multiple binary operations one after the other, like what happens inside of `Stream.reduce()`, you end up with tons of different ways of grouping those calls.

Listing 7.20 Calling a binary operation with different groupings

```
RawData row1 = new RawData(...);  
RawData row2 = new RawData(...);  
RawData row3 = new RawData(...);  
  
add(add(row1, row2), row3); #A  
add(row1, add(row2, row3)); #B
```

#A First we add rows 1 and 2, then add that result to row 3

#B or we could first add rows 2 and 3, and then add that result to row 1

These call styles can lead to wildly different results depending on the underlying implementation. For instance, if we're performing subtraction, then any change in argument order or grouping can cause the same set of inputs to produce completely different outputs.

Listing 7.21 Same inputs, different answers

```
int x = 1;  
int y = 2;  
int z = 3;  
x - y; #A  
y - x; #A  
1 - (2 - 3) = 2 #B  
(1 - 2) - 3 = -4 #B
```

#A The order in which we apply the operations changes the answer

#B similarly, different groupings give different answers

Even pedestrian operations like addition can lead to wacky results if the underlying data type doesn't offer the properties you expect.

Listing 7.22 Adding floating point numbers is *not* an associative binary operation

```
double a = 1.00001;  
double b = 1.00002;  
double c = 1.00003;  
  
Assertions.assertEquals( // FALSE! #A  
    (a + (b + c)), #A  
    ((a + b) + c) #A  
);
```

#A Dang you, floating point!

The subtlety goes beyond just argument order. There can be *relationships* between inputs and outputs, as well as expectations around how much information an operation should "carry through" between its inputs and outputs.

Listing 7.23 Algebraic operations can sometimes be governed by special laws

```
double x = 0.02
double y = 0.02
assertTrue(x == someBinaryOperation(x, y) == y);    #A

assertEquals(                                #B
    x <= y,                                #B
    someUnaryOperation(x) <= someUnaryOperation(y) #B
)
```

#A A perfectly normal expectation for some binary operations!

#B Sometimes we want an algebraic operation to maintain some aspect of the input values. For instance, if the two inputs are sorted before we apply the operation, we might want their output to “keep” that same ordering

How an algebraic operation behaves under various orderings and groupings and how it relates to its inputs and outputs are what we call its *properties*. They define what it means for an algebraic operation to be correct.

And that brings us to a quick detour: how to describe what correct *means*.

7.5 Invariants, Properties, State, and an upside-down capital A

Algebraic properties are built around a simple style of assertion that goes like this: “for all possible states, [some invariant] always holds.”

It looks like this in Java.

Listing 7.24 Algebraic style assertions in Java

```
for (State state : allPossibleStates()) {
    assertTrue(someInvariant(state));
}
```

What do we mean by “all possible states”? This is where the denotational powers we explored in previous chapters come back into play. When we think about what a type like Integer “means” in Java, we learned to see it as being made up of all the individual whole numbers that 32 bits can represent. These individual values make up every possible state an Integer could be in our program.

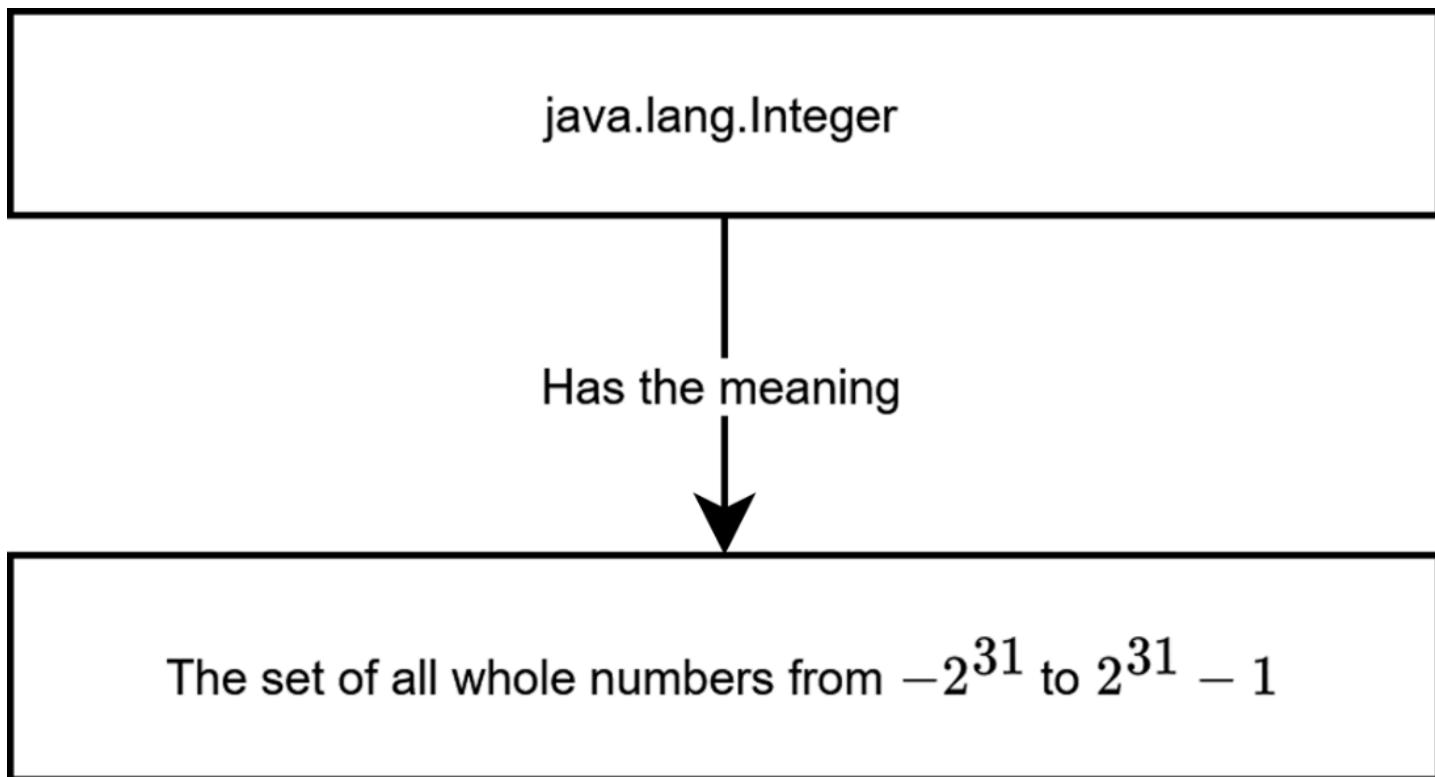


Figure 7.15 The set of values that comprises an Integer in Java

So far, we've used these denotations primarily as a design tool. We often pick data types in Java that denote sets of values that don't line up with our intentions. Thinking through the meaning of our types was a quick and easy way to notice this mismatch and correct our design error.

Now we're going to take it one step further. We're not going to just *think* about the set of values that our types denote, we're going *create* them and use them to make assertions! If we're going to reason algebraically about invariants that "hold for all states," we have to be able to say what those states *are*.

This is really easy to do when data types are simple. For instance, we know the set of values that a Java Integer denotes. Generating every single one of them only needs a for-loop going from the smallest value to the largest.

Listing 7.25 asserting that an invariant holds for all possible values of an integer

```

for (int i = Integer.MIN_VALUE; i<Integer.MAX_VALUE; i++) { #A
    assertEquals(i + i, 2 * i); #B
}
  
```

#A Our main design phrase: "For all possible Integers, it holds that..."

#B Interesting, this law holds even as we overflow and wrap around, which is pretty neat

Figuring out all of the states in a composite data type like a record is no different from doing it for individual data types like Integer. It just takes a bit of careful tallying. In fact, the algebraic name for records, *Product Types*, tells us exactly how to do it! You figure out the set of values that each attribute denotes, then stitch those all together as a cartesian product.

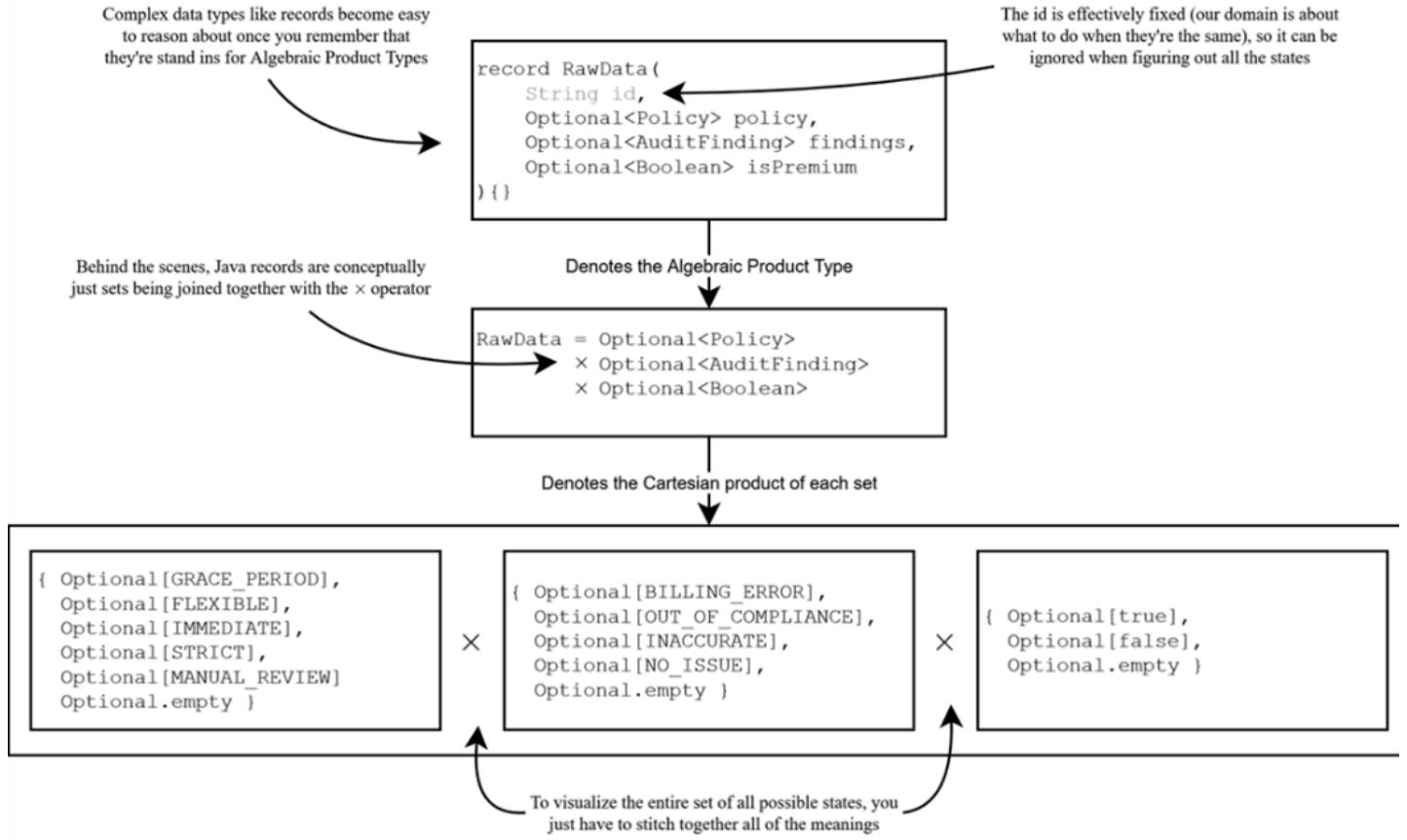


Figure 7.16 figuring out the concrete set of values that complex data types denote

Translating this into Java is equally simple. We just have to make sure we remember to count the extra empty state that comes from the Optionals.

Listing 7.26 Every possible state that RawData could represent

```
static Set<RawData> everyPossibleRow() { #A
    Set<RawData> output = new HashSet<>();
    for (var policy : everyOptionalValue(Policy.values())) {
        for (var finding : everyOptionalValue(AuditFinding.values())) {
            for (var premium : everyOptionalValue(List.of(true, false))) {
                output.add(new rawData(
                    "FixedCustomerId",
                    policy,
                    finding,
                    premium)
            );
        }
    }
    return output;
}

@SafeVarargs
static <A> Set<Optional<A>> everyOptionalValue(A... items) { #B
    return Stream.concat(
        Stream.of(items).map(Optional::of),
        Stream.of(Optional.<A>empty())
    ).toSet();
}
```

#A This generates every possible state that our RawData type could be for an individual customer ID.
#B This little helper makes sure every Optional value (including empty) gets accounted for

And once we have that set of data, we can use it to make powerful assertions that exhaustively prove what's under test for every possible state.

Listing 7.27 An exhaustive assertion against every possible row

```
For (RawData row : everyPossibleRow()) {
    assertTrue(someInvariant(row));
}
```

Now comes the really cool part. We can take this exhaustive set of every possible value and combine it with *another* exhaustive set of every possible value!

Listing 7.28 Combining the same set of states

```
for (RawData a : everyPossibleRow()) {      #A
    for (RawData b : everyPossibleRow()) { #A
        // assert something about a and b   #B
    }
}
```

#A These two loops allow us to explore every single pairing of possible states
#B And make assertions about them!

This is useful because it expands our “for all” style reasoning into a world where we can talk about *relationships* between states. “For any two states, a and b, out of all the possible states, we expect [some invariant] to hold between them.”

Our algebra happens to be all about what happens when we smoosh two states together. This style of assertion lets us explore every possible way in which data could

be combined.

Listing 7.29 How we'll make assertions about our algebra

```
for (RawData a : everyPossibleRow()) {  
    for (RawData b : everyPossibleRow()) {  
        assertEquals(  
            add(a, b),  #A  
            ???         #B  
        );  
    }  
}
```

#A we can make assertions about the behavior of our algebra 'for all' possible states
#B Of course, we don't know what those properties are yet, but we're getting closer

There's no limit to this combining. We can combine the same set with itself repeatedly to create assertions about larger and larger relationships between states.

Listing 7.30 States can be recombined over and over into larger sets

```
for (RawData a : everyPossibleRow()) {  
    for (RawData b : everyPossibleRow()) {  
        for (RawData c : everyPossibleRow()) {  
            // assert something about a and b and c  
        }  
    }  
}  
// or 4 or 5 or more!
```

This is another item in that growing set of “obvious, but somehow not so obvious” things we keep exploring. It’s very common for programmers to think about individual examples. Maybe even sets of examples through something like JUnit’s `@ParameterizedTest`, but for some reason, we usually don’t think about *every possible* example. I programmed for a long time before having an “oh, wait – I could totally check all of them” style epiphany. This small leap from single examples to “for all” examples opens up an entirely new world of thinking about correctness in our programs. A very different kind of confidence can be drawn from a test suite that leaves no possible state unexplored.

7.5.1 Every possible state? Surely too inefficient to be practical!

Java programmers love to fret about performance. So, let’s tally up every state in our domain.

Listing 7.31 A quick reminder of the main data types

```
public enum Policy {  
    GRACE_PERIOD, FLEXIBLE, IMMEDIATE, STRICT, MANUAL_REVIEW  
}  
  
public enum AuditFinding {  
    BILLING_ERROR, OUT_OF_COMPLIANCE, INACCURATE, NO_ISSUE  
}  
  
record RawData(  
    String id,  
    Optional<Policy> policy,  
    Optional<AuditFinding> findings,  
    Optional<Boolean> isPremium  
){}  

```

The Policy enum has 5 values. AuditFinding has 4 and Boolean has 2. Each are wrapped in an Optional, so we've got to count their extra `empty()` state. That brings the grand total to... $6 \times 5 \times 3 = 90$ unique possible states.

That's not a lot.

Computers are really good at generating data really quickly. This naïve approach of making all the states by hand with for-loops and elbow grease works for a surprisingly large scope of problems. Most state spaces, if you take the time to model them well, aren't that large. A few million objects in memory is nothing for the JVM.

As with all performance worries, when in doubt: benchmark. Run those for-loops from Listing 7.30 above. They'll generate just shy of 1 million 3-element tuples. On my dusty old Desktop (still rocking HDDs), it takes all of 200ms.

Of course, there are plenty of times where this won't be computationally possible, not worth it, or just too annoying to do by hand. The good news is that there's a rich world of property-based testing libraries which can generate data for us. The bad news is that they're not that good in Java. But that's a story for our chapter on testing.

For now, we'll stick with for-loops.

7.5.2 Expressing these ideas without code

This "for all possible states" expression is so common that it has a shorthand we can steal from the math people. It looks like this: \forall . Literally an upside-down capital A. When you see that, read it as "for all."

Once we say "for all," we have to say "for all *what*." So, we pair this upside-down "A" with a variable, like `x`, and a set that describes all the states, like `Rows`. That gives us a way of saying something really sophisticated in a compact way.

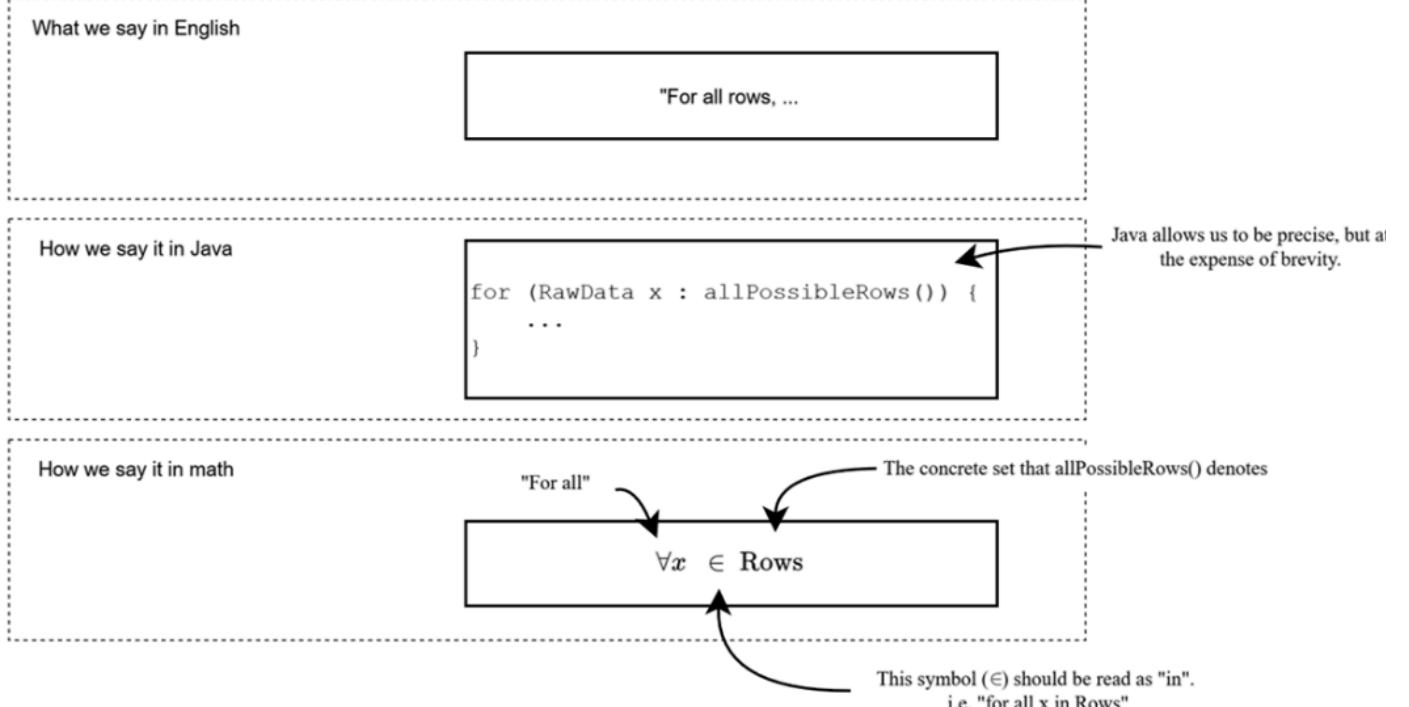


Figure 7.17 Translating between English, Java, and math

This terse syntax is super valuable for talking about invariants. It introduces a level of precision that the English prose version lacks. It specifies that we're talking about a *set* of values, and that we're using a variable, x , to denote an individual value in that set, and, thanks to the \forall quantifier, that we're talking about every single one of those values. All of that in a few symbols!

Of course, you could accomplish the same with Java, but it comes with all the baggage of for-loops, boilerplate, and needing to define the iterable sets by hand. All of that gets in the way of what we're trying to accomplish at this stage in the game: talk about what we're doing. Having a tiny "language" where we can express things succinctly frees us to focus on the thinking part without getting bogged down in code.

The succinctness really pays off when we start talking about algebraic properties that involve relationships between states (like our addition operation). Doing it with our shorthand looks like this:

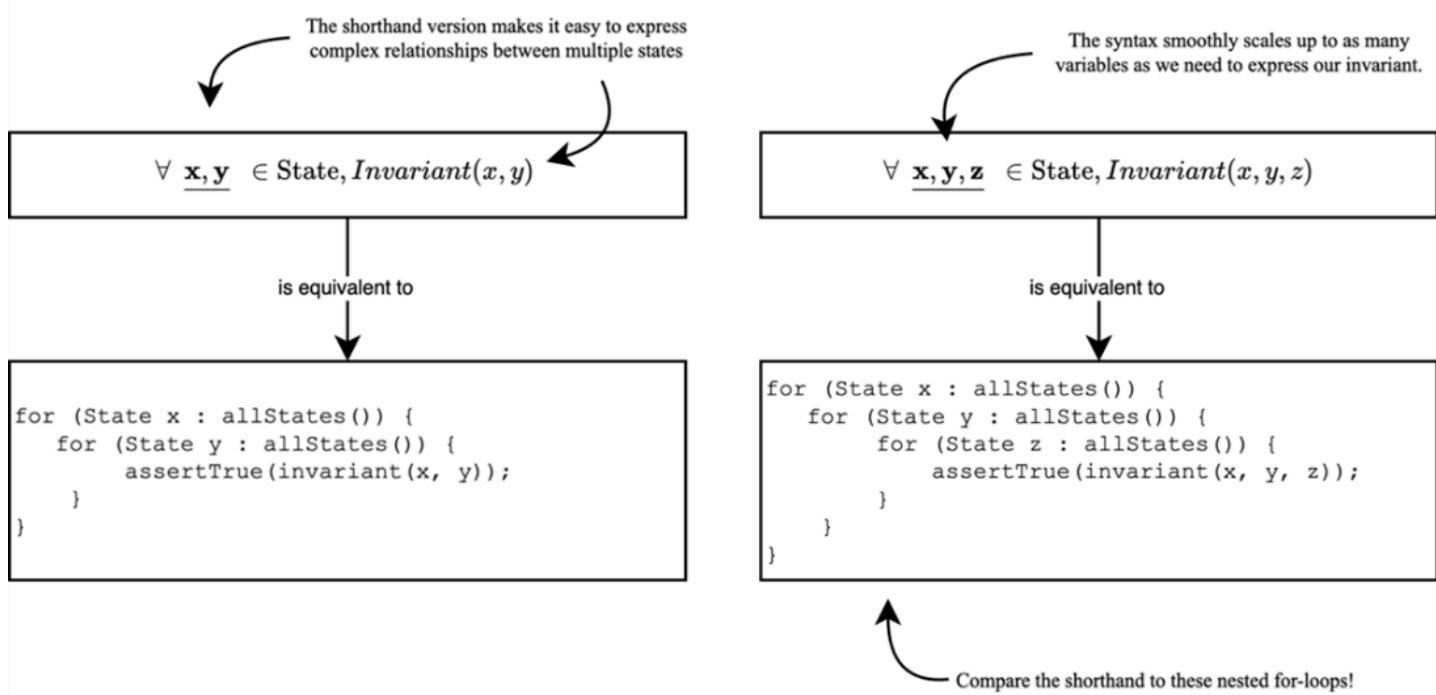


Figure 7.18 Stealing from the math people lets us express complex ideas succinctly

The usefulness of this notation goes beyond being shorter than Java or English. The act of writing out “for all” forces us to change how we think about a problem. What *are* the states? Can we enumerate them? Which invariants *should* hold for them? Formal notation is a short cut towards realizing how sloppy our thinking is most of the time.

So, throughout the rest of this chapter we’re going to force ourselves to write down what we know using this formal “for all” notation. It’ll start out vague, but we’ll refine it as we go.

Here’s a first stab at our binary operation:

Listing 7.32 Writing down what we know

$\forall a, b \in \text{ConflictingRows}$

Properties:

$a + b = "a \text{ more favorable record}"$ (whatever that means) #A

#A This isn’t formal yet, but it’s a start

Of course, this definition isn’t usable yet, but it’s a step in the right direction.

To go further, we have to expand our vocabulary of algebraic properties. There are shoulders to stand on. Let’s explore the properties that power our binary operations.

7.6 Associativity

Associativity describes how binary operations should behave when grouped. It’s a way of saying that “where we put the parenthesis” doesn’t matter.

It looks like this using our shorthand notation:

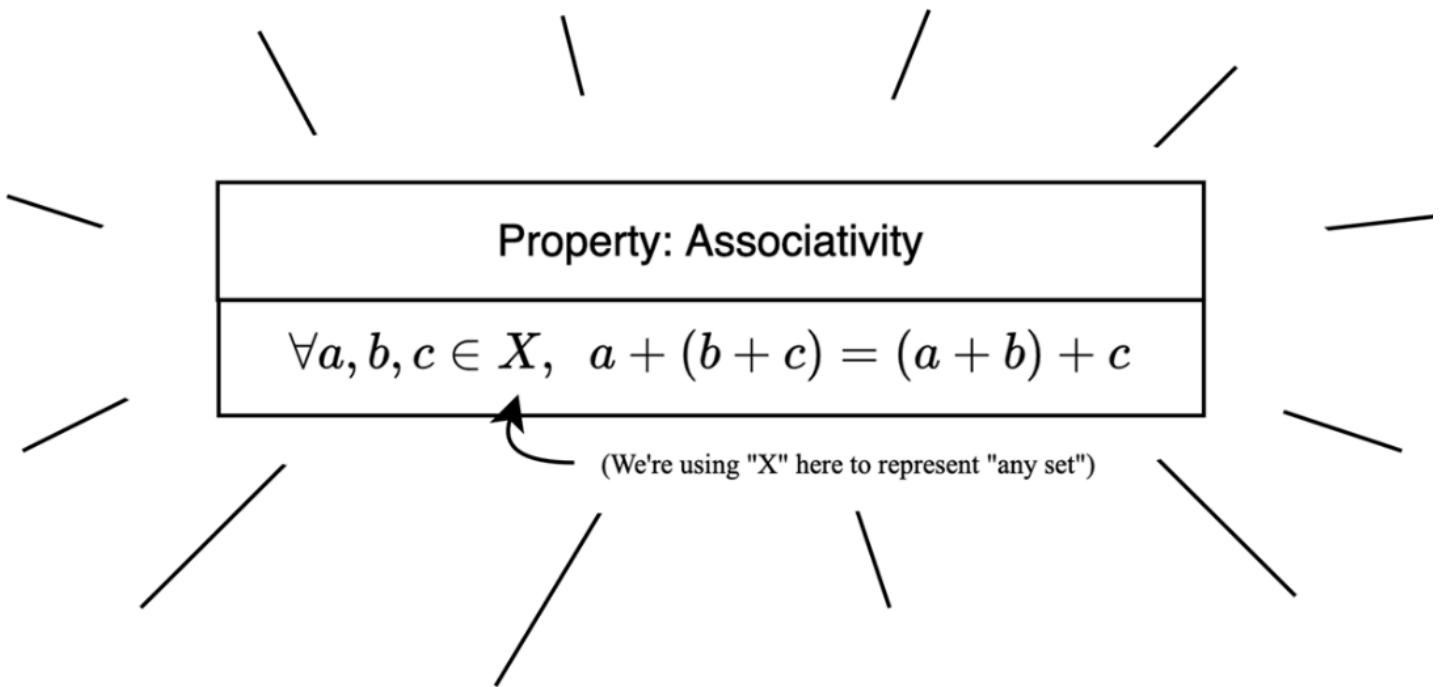


Figure 7.19 The associativity property

There are two sides to associativity worth exploring. The first is just from a “well, duh. Of course, it should work that way” perspective. Associativity is so intuitive and “obvious” that we have to force ourselves to notice it.

Listing 7.33 “of course” all of these should give the same answer

```
RawData row1 = new RawData(...);
RawData row2 = new RawData(...);
RawData row3 = new RawData(...);
assertEquals(
    add(row1, add(row2, row3)), #A
    add(add(row1, row2), row3) #A
)
```

#A It would be straight up weird if these led to different answers

The second is much grander in scope and captures the soul of what makes algebraic operations beautiful. Associativity means we don’t have to care about who does the grouping – or *where* they do it.

Associativity allows us to break apart any sequence of operations into arbitrary chunks

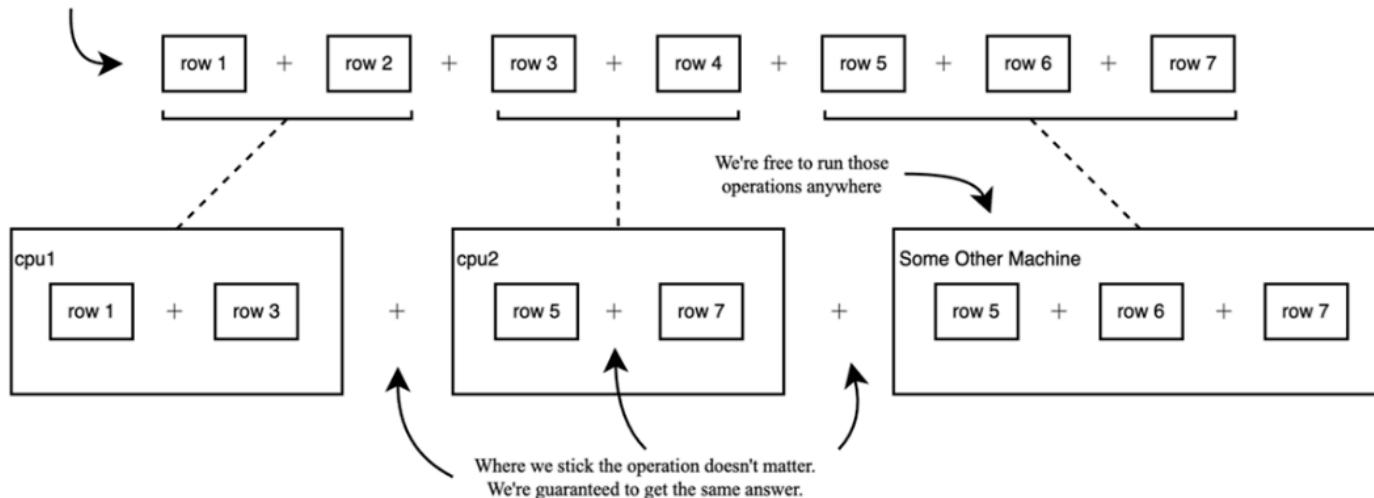


Figure 7.20 Associativity buys flexibility in execution

Associativity is *why* we can get rid of for-loops and hand everything over to the Streams API to let reduce figure out the best way to perform the work – including potentially in parallel.

Listing 7.34 Associativity is the bedrock property of Stream parallelism

```
static <K, V> Map<K, V> toMap(
    List<V> items,
    Function<V, K> classifier,
    BinaryOperator<V> binop
) {
    return items.stream()
        .parallel() #A
        .map(item -> Map.of(classifier.apply(item), item))
        .reduce(Map.of(), (m1, m2) -> add(m1, m2, binop));
}
```

#A Associativity is a requirement for correct parallel stream processing

Most valuable of all, this property tells us something about what a correct implementation *should* be. It's a concrete law we can write down.

Listing 7.35 Writing down what we know about our algebra's associativity

$\forall a, b, c \in \text{ConflictingRows}$

Properties:

$a + b = "a \text{ more favorable record}"$ (whatever that means)
Associative: $(a + (b + c)) = ((a + b) + c)$ #A

#A The new law that any correct implementation must satisfy

And now that we know what correct means, we can write a test for it!

Listing 7.36 Verifying associativity with an exhaustive unit test

```
void testAssociativity() {  
    for (RawData a : allPossibleRows()) {      #A  
        for (RawData b : allPossibleRows()) {      #A  
            for (RawData c : allPossibleRows()) { #A  
                assertEquals(                      #B  
                    add(a, add(b, c)),               #B  
                    add(add(a, b), c));          #B  
            }  
        }  
    }  
}
```

#A Translates to “ $\forall a, b, c \in \text{RawData}$ ”

#B “The associativity laws must hold”

7.7 Noticing more properties

Let's look at the main binary operation that combines our RawData types. To make analyzing the code easier, we'll temporarily ignore the optional/null side of the requirements. Instead, we'll write our implementation as though the data is always available.

Here's a refresher on what our implementation needs to do.

Table 7.2 A refresher on the requirements

Replace nulls with concrete values (We'll skip this one for now)
If two values conflict, pick the one that favors not billing the customer
If two values have the same customer impact, resolve via sort

Let's translate these into code.

Listing 7.37 Implementing binary operations for each individual type

```
static RawData add(RawData x, RawData y) {      #A
    if (!x.id().equals(y.id())) {                #B
        throw new IllegalArgumentException(      #B
            "Hey! Only conflicting rows allowed!" #B
        );
    }                                              #B
    return new RawData(
        x.id(),
        add(x.policy(), y.policy()),           #C
        add(x.findings(), y.findings()),       #C
        add(x.premium(), y.premium())          #C
    );
}

static Policy add(Policy x, Policy y) {
    if (policyImpact(x).equals(policyImpact(y))) { #D
        return x.name().compareTo(y.name()) >= 0 ? x : y;  #E
    } else {
        return policyImpact(x).equals(CustomerImpact.FAVORS) ? x : y;  #F
    }
}
static AuditFinding add(AuditFinding x, AuditFinding y) { #G
    if (findingsImpact(x).equals(findingsImpact(y))) {
        return x.name().compareTo(y.name()) >= 0 ? x : y;
    } else {
        return findingsImpact(x).equals(CustomerImpact.FAVORS) ? x : y;
    }
}
static Boolean add(Boolean x, Boolean y) {
    return x || y;    #H
}
```

#A Here's the main binary operation that combines our data
#B The algebra is only defined for records which conflict, thus the defense here
#C Algebraic approaches produce these pleasing, uniform top-level APIs
#D First we check to see if they have the same customer impact (req. A2)
#E If they do, we resolve by an arbitrary lexicographical sort (req. A3)
#F Otherwise, we take whichever one is more favorable (req. A1)
#G Here we perform the exact same logic.
#H Booleans are easy as pie. Take whichever is true (i.e. favors the customer by making them a "premium" account)

There are a few things to notice with this implementation. This first is how complex some of it is. The functions for adding Policy and AuditFinding are a punishing mix of if statements and ternaries. The second is that how much of it is duplicated. DRY alarms are probably firing in your head.

Do we refactor? The question would be to *what*?

If we go the DRY route, we might extract the duplicated handling logic into its own function and then delegate the original calls.

We could DRY the implementation by making it work with any Enum, rather than a specific one

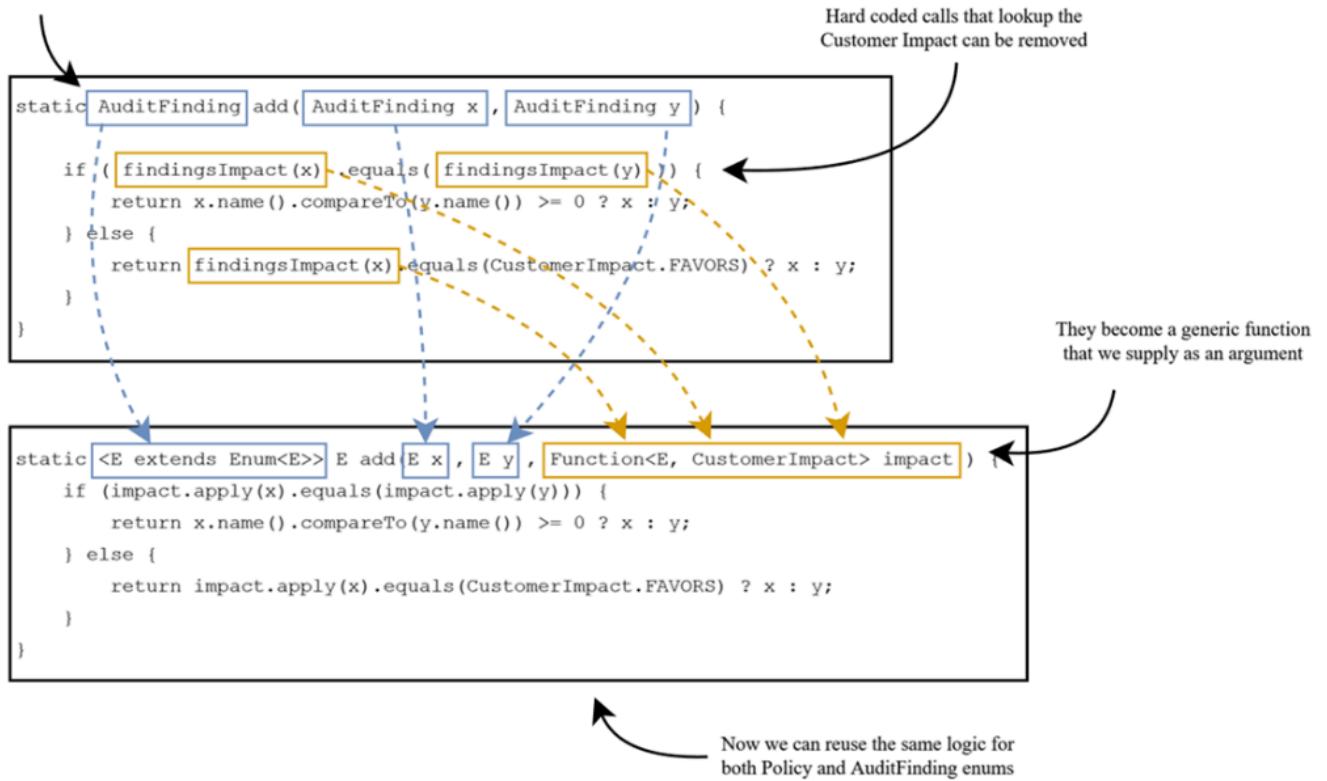


Figure 7.21 DRYing the code by extracting the duplicated logic into its own function

Then we could DRY the rest of the implementation, too.

Listing 7.38 DRY == Better?

```

static <E extends Enum<E>> E add(E x, E y, Function<E, CustomerImpact> f) {
    if (f.apply(x).equals(f.apply(y))) {
        return x.name().compareTo(y.name()) >= 0 ? x : y;
    } else {
        return f.apply(x).equals(CustomerImpact.FAVORS) ? x : y;
    }
}

static Policy add(Policy x, Policy y) {
    return add(x, y, Starting::policyImpact);      #A
}
static AuditFinding add(AuditFinding x, AuditFinding y) {
    return add(x, y, Starting::findingsImpact);   #A
}
static Boolean add(Boolean x, Boolean y) {
    return x || y;
}

```

#A Replacing the duplicated logic with our new function

It's smaller, and DRYer, but... is it better?

What about "simpler"?

The original questions remain even after the refactor: how would you test this code? What does each conditional branch *mean*? How will you know its correct? Any tests we write for this are going to be annoyingly coupled to *how* the code makes a particular choice.

This refactor made the code prettier, but it didn't solve the underlying problem: we're missing something -- some unifying insight that tells us what the code is supposed to do. To refactor from here, we have to dig down to the properties hidden below the surface.

So, let's take a step back from the code and look at the requirements again.



Figure 7.22 Another look at the requirements

What the requirements are really saying is that favorability is *measurable*; It's a unit that can be *compared*.

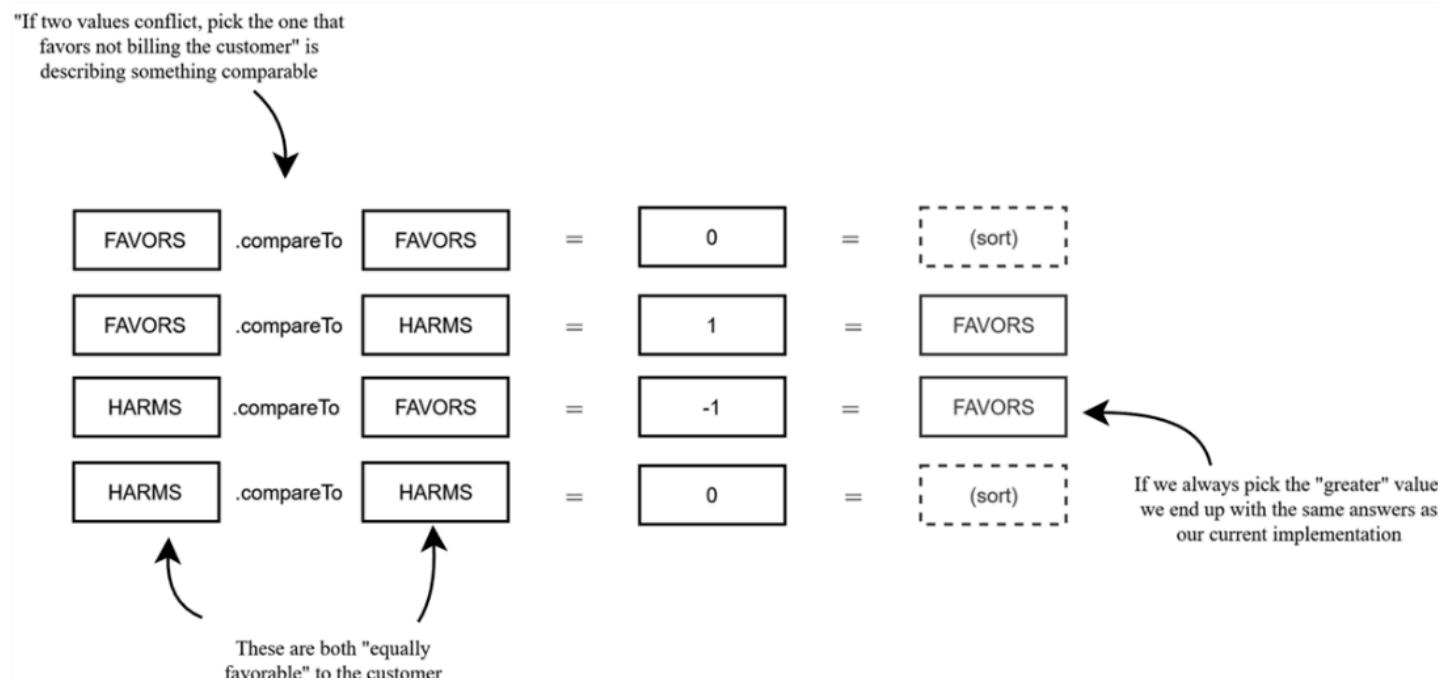


Figure 7.23 Describing the requirements in terms of Comparator

This being a sorting problem is probably painfully obvious once it's pointed out (our requirements even have the word "sort" in them), but it can be pretty challenging in practice to realize that an arbitrary set of requirements is sortable – especially when they're wrapped up in other concerns like handling null/optional values.

You'd think recognizing when things are ordered would be second nature to developers. You surely know (or have forgotten) numerous sorting algorithms.

However, that familiarity is exactly what leads to blind spots. Sorting is usually about *values*. We sort by times, ages, prices, volumes, sales, and so on. But orders can also be about *ideas*. Many of which can defy our intuition around "sortability." Customer "favorability" isn't an explicit value in our code; it's an *idea* about what those values *mean*.

We can define an ordering the same way we define an algebra. We stick a symbol (\leq , by convention) between two pieces of data, and then ask our favorite question: what should that *mean*?

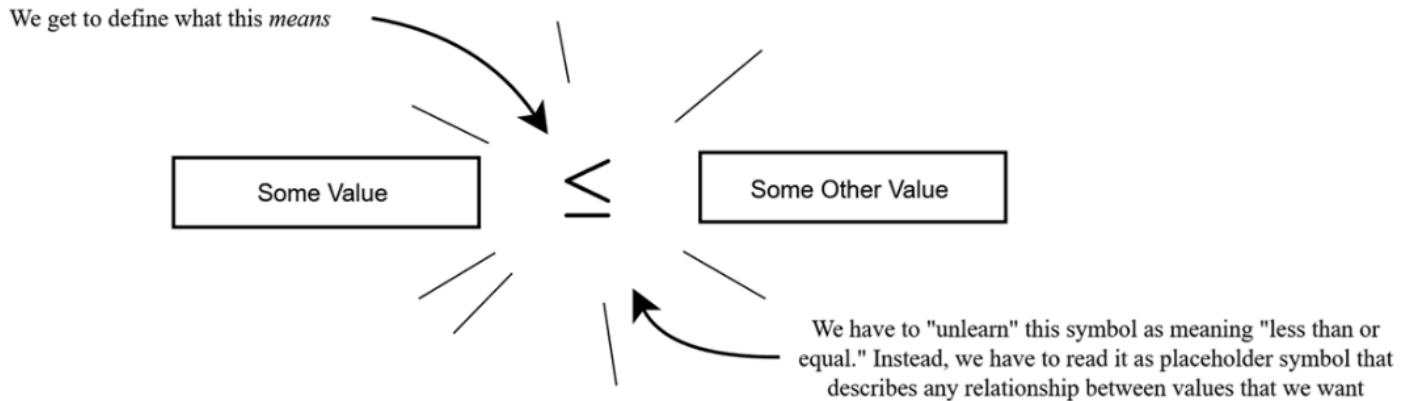


Figure 7.24 We get to decide what the relationship between our values means

We can define that meaning by starting at the top with `RawData` and working our way down to each of its attributes.

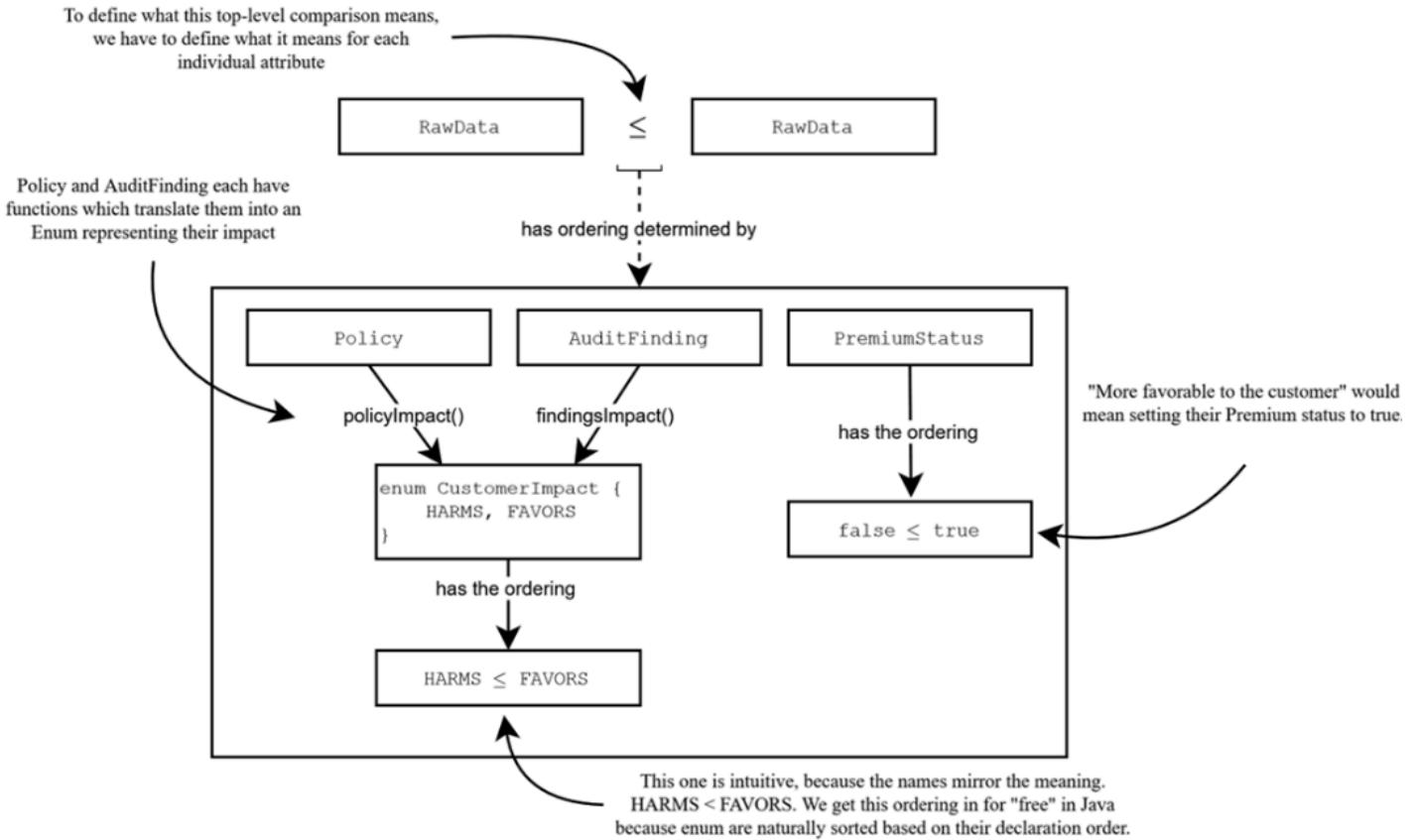


Figure 7.25 Defining what it means for one record to be “more favorable” than another

We can be similarly explicit about how to handle collisions. Our requirements only specify that *some* “sort” takes place; we can firm that up to say we’ll specifically sort lexicographically.

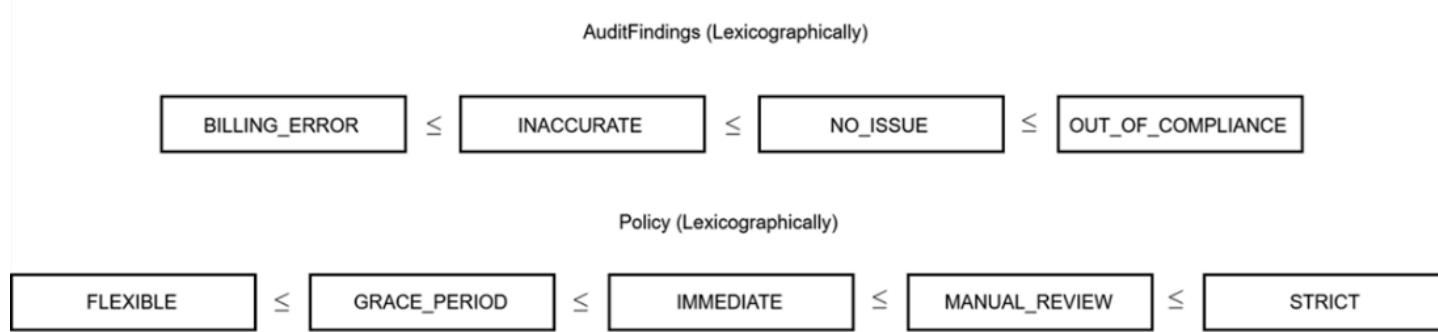


Figure 7.26 Policy and AuditFindings sorted lexicographically

The really nice part of realizing your problem is just a sorting problem is then realizing that it’s a *solved* problem. We can offload all the work to Java’s build in Comparator tooling.

Listing 7.39 Refactoring to express our logic in terms of ordering

```

static Policy add(Policy x, Policy y) {
    Comparator<Policy> comparator = #A
        comparing(Chapter7::policyImpact) #A
            .thenComparing(Policy::name); #A
    return comparator.compare(x, y) > 0 ? y : x;
}

static AuditFinding add(AuditFinding x, AuditFinding y) { #B
    Comparator<AuditFinding> comparator = #B
        comparing(Chapter7::findingsImpact) #B
            .thenComparing(AuditFinding::name); #B
    return comparator.compare(a, b) > 0 ? y : x;
}

static Boolean add(Boolean x, Boolean y) { #C
    // Boolean.compare(x, y) > 0 ? x : y;
    return a || b
}

```

#A Expressing our problem in terms of Comparator

#B Our refactor has about the same number of lines of code, but our implementation is reading much more like our requirements. "Sort by favorability, and then name"

#C While we could express the "more favorable" on Booleans in terms of a comparator, it's hard to beat the familiarity of logical OR.

Line for line, it's about the same as the super-DRY refactor from before, but it's doing something more interesting. We've begun to shape the implementation around the properties that power it.

7.7.1 Ordering as a property

The biggest value in noticing that your domain can be described by an ordering has little to do with the specific implementation and everything to do with what it allows us to say about correctness. Orderings are backed by a set of extremely useful properties.

They look just like the equality properties we covered back in Chapter 2.

Property: Reflexivity	$\forall a \in X, a \leq a$
Property: Antisymmetry	$\forall a, b \in X, \text{ IF } a \leq b \text{ AND } b \leq a \text{ THEN } a = b$
Property: Transitivity	$\forall a, b, c \in X, \text{ IF } a \leq b \text{ AND } b \leq c \text{ THEN } a \leq c$

Figure 7.27 The properties associated with Java's standard Comparable interface

Total orders, like Comparable, need all three of these properties. However, in general, in an ordering could be made up of just one of them. Different relationships will have

different subsets. For instance, if we define \leq to mean “is parent of,” then our order will surely *not* be reflexive (I can’t be my own parent). Similarly, if modeling friendship, we might lose transitivity (my friend’s friend is not necessarily *my* friend).

This flexibility allows Orders to represent amazingly complex ideas that go far outside of the usual Comparator notion we’re familiar with. They can model questions about capability (“is this permission sufficient to do what I want?”), creation (“given eggs and butter, can I create an omelet?”), hierarchy (“is Jane a parent of Bob?”), classification (is “tiger” more specific than “mammal”?), or any other arbitrary relationship we can imagine.

In our domain, we’ve expressed “favorability” as a total order that’s reflexive, transitive, and antisymmetric.

We can write this down

Listing 7.40 Further refining what correct means in our domain

```
a,b,c,d      ConflictingRows

Properties:
  Associativity: (a + (b + c)) = ((a + b) + c)
  Ordered:
    Reflexivity: a ≤ a          #A
    Antisymmetry: IF a ≤ b AND b ≤ a THEN a = b #A
    Transitivity: IF a ≤ b AND b ≤ c THEN a ≤ c #A
```

#A Writing down what we know about our data’s ordering

And we can use the knowledge of these properties to write tests that sit “above” the implementation. The properties free us from caring about what a specific branch or ternary does in some specific function. If we exercise all states, and the properties hold for all of them, then the underlying implementation *must* be correct.

Listing 7.41 Verifying the low-level properties that power Comparator

```
Comparator<AuditFinding> comparator =          #A
    comparing(Chapter7::findingsImpact)      #A
        .thenComparing(AuditFinding::name); #A

void testReflexivity() {
    for (AuditFinding a : AuditFinding.values()) {
        assertTrue(comparator.compare(a, a) <= 0)    #B
    }
}

void testAntisymmetry() {
    for (AuditFinding a : AuditFinding.values()) {
        for (AuditFinding b : AuditFinding.values()) {
            if (comparator.compare(a, b) <= 0
                && comparator.compareTo(b, a) <= 0) { #C
                assertEquals(a, b);                  #C
            }
        }
    }
}

void testTransitivity() {
    for (AuditFinding a : AuditFinding.values()) {
        for (AuditFinding b : AuditFinding.values()) {
            for (AuditFinding c : AuditFinding.values()) {
                if (comparator.compare(a, b) <= 0
                    && comparator.compare(b, c) <= 0) {           #D
                    assertTrue(comparator.compare(a, c) <= 0); #D
                }
            }
        }
    }
}
```

#A We'll just look at just one of the data types for ease of example.

#B Reflexivity might seem like a pointless thing to assert, but it's possible to be violated if your implementation has customized its comparator implementation

#C Verifying antisymmetry holds. This is another one that seems too obvious to bother with, but it actually caught a bug in my original implementation while working on this chapter!

#D Verifying that transitivity holds

Breaking out each property like this is useful in a book, but in practice, a much nicer way to test orderings is to generate every possible permutation of your data set, and then verify that every possible input leads to a single sorted output.

Listing 7.42 verifying ordering properties by checking all permutations

```
@Test
void anExerciseForTheReader() {
    List<AuditFinding> expectedOrder = List.of(
        INACCURATE, NO_ISSUE,
        BILLING_ERROR, OUT_OF_COMPLIANCE);

    assertTrue(
        permutations(AuditFinding.values()) #A
            .stream()
            .map(x -> x.stream().sorted().toList())
            .allMatch(sorted -> sorted.equals(expectedOrder))); #B
}
```

#A The implementation of this permutation function is left as an exercise

#B If every possible input permutation produces the same sorted output, we've proven our ordering properties hold

If this property holds all the other ones do, too.

7.8 Monotonic Relationships between inputs and outputs

Our algebra has one job: pick the “more favorable” option when combining data. Once a new level of “favorability” is achieved, it should never regress.

This behavior seems like it should come for free since our data is ordered, but there are lots of ways to break that ordering once you start chaining operations together.

For instance, let’s look at the integers. They’re totally ordered by \leq , but subtraction and multiplication can cause the relative order *between* the values to flip. Somehow, adding two “smaller” values can cause a “bigger” value to emerge – one that’s even bigger than adding two “big” values might produce on their own!

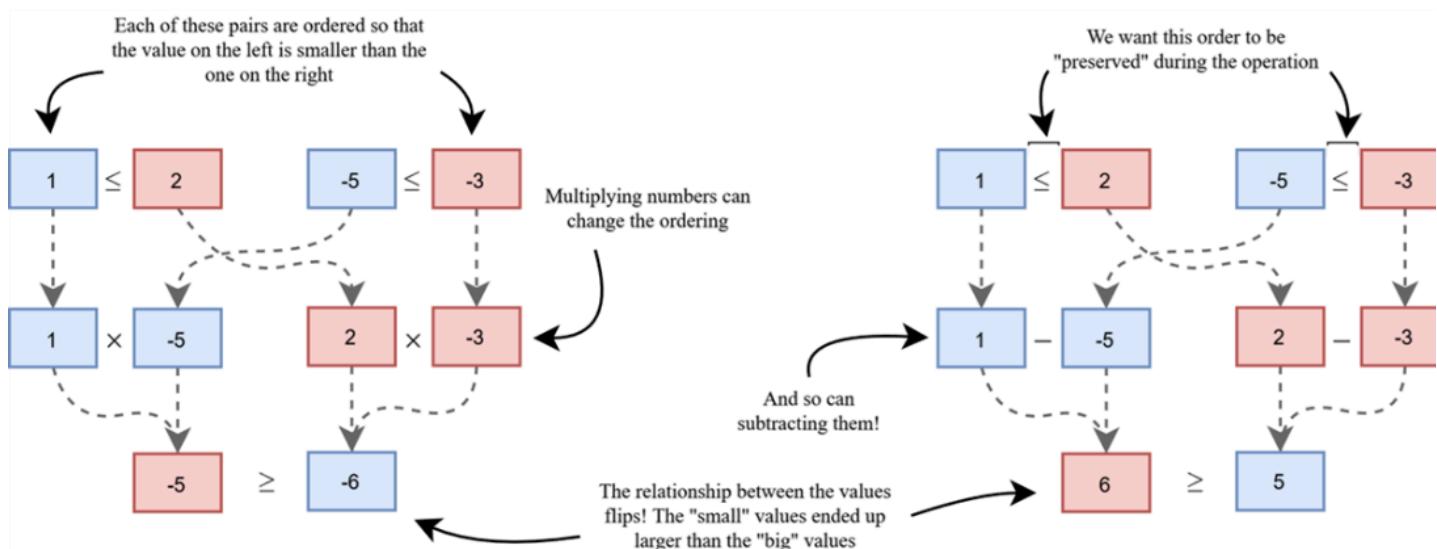


Figure 7.28 Examples of order not being preserved during an operation

A faulty implementation could cause this to happen in our domain even though the data itself is totally ordered.

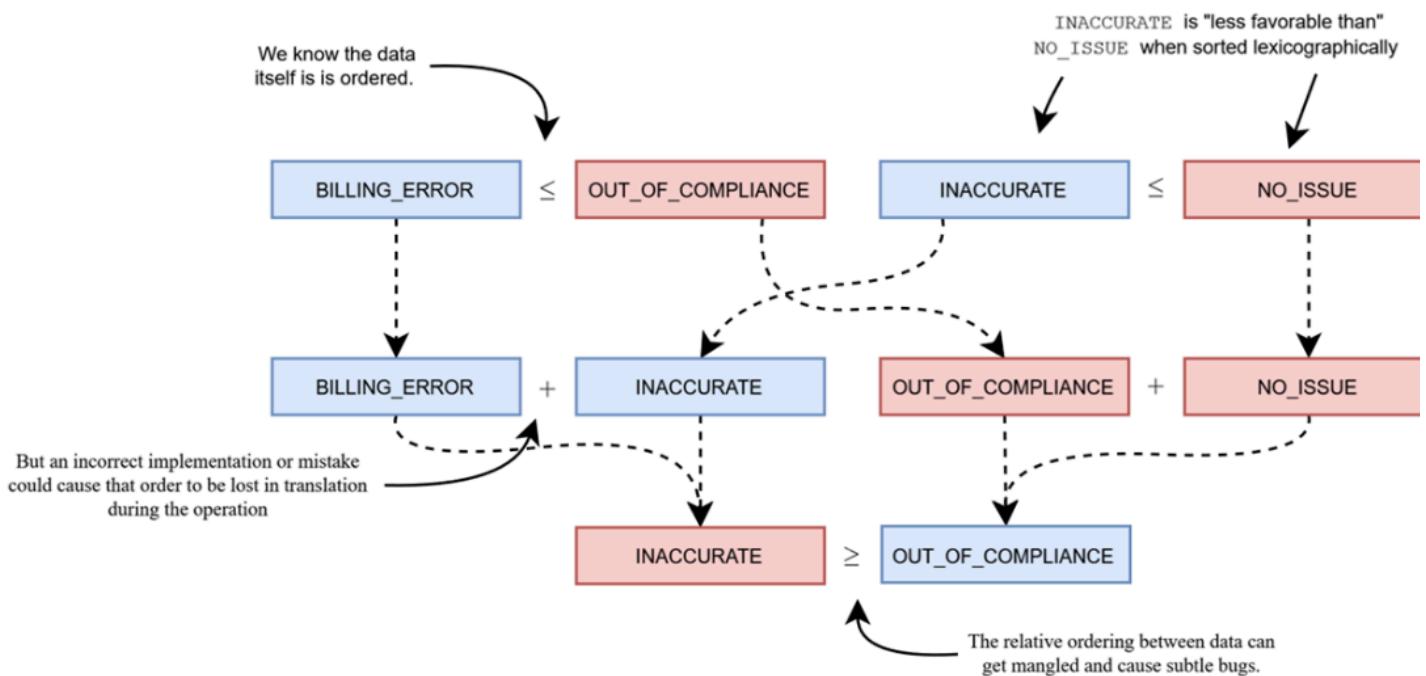


Figure 7.29 Losing order while adding two data types together

So, ordering on its own isn't enough to fully say what it means for our implementation to be correct. We have to be able to say that the operations in our domain *preserve* the data's ordering.

This leads us to the final property we'll explore: *monotonicity*

It looks like this in our short hand:

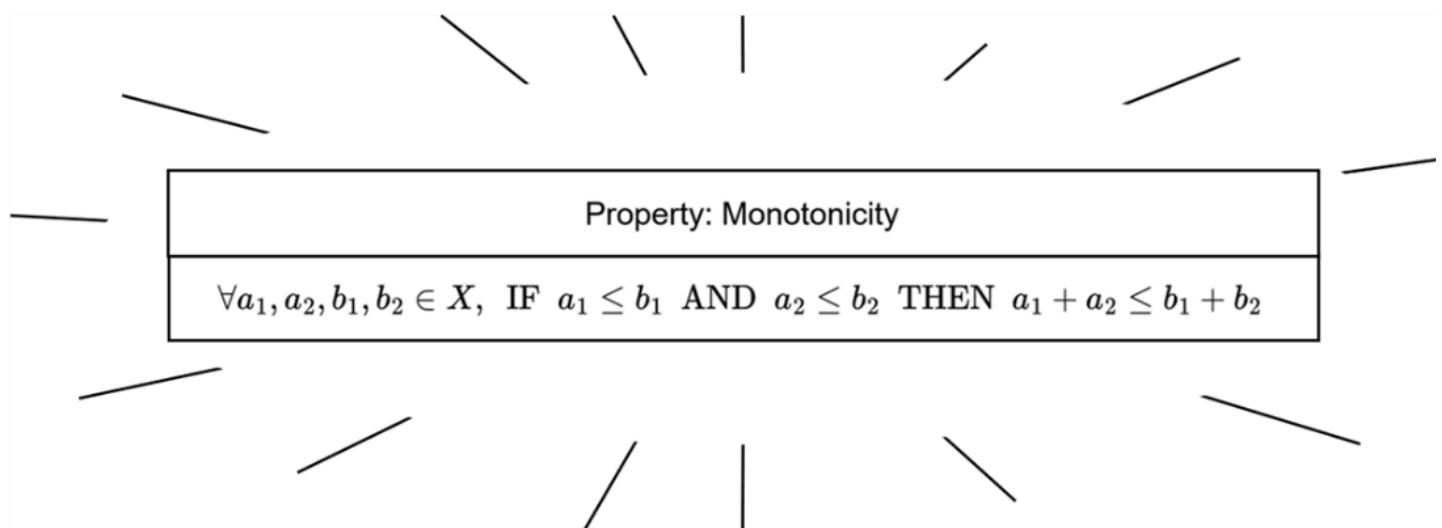


Figure 7.30 The Monotonic property

Scary looking, right? This property looks complex, and thus seems like it should be saying something complex, but it's saying something simple and obvious. Our operation should keep "big" things big and "small" things small.

But all of those variables make it look tricky. So, let's translate this into a test in Java to convince ourselves that everything here is reasonable.

Listing 7.43 Translating the monotonic law into Java

```
@Test
void testMonotonicity () {
    for (RawData a1 : allPossibleRows()) {          #A
        for (RawData a2 : allPossibleRows()) {          #A
            for (RawData b1 : allPossibleRows()) {      #A
                for (RawData b2 : allPossibleRows()) {  #A
                    if (a1.compareTo(b1) <= 0 && a2.compareTo(b2) <= 0) {
                        assertTrue(
                            add(a1, a2).compareTo(add(b1, b2)) <= 0  #B
                        );
                    }
                }
            }
        }
    }
}
```

#A We could also name these variables as a,b,c,d like we've done in previous examples, but this labeling tends to make the relationships a bit clearer

#B The comparison between these operations is key. It verifies that no matter what we feed in as arguments, we only move "forward" in terms of favorability.

It might still feel weird and foreign, but it's worth familiarizing yourself with. Monotonic functions are *everywhere* in programming. Any time your domain needs to tighten, constrain, or refine, you're likely dealing with something that can be described by a monotonic function. Recognizing it gives you access to their laws.

The comparison in our implementation has exactly this monotonic behavior.

Listing 7.44 What makes our functions monotonic

```
static Policy add(Policy x, Policy y) {
    Comparator<Policy> comparator =
        comparing(Chapter7::policyImpact)
            .thenComparing(Policy::name);
    return comparator.compare(x, y) > 0 ? y : x;           #A
}
static AuditFinding add(AuditFinding x, AuditFinding y) {
    Comparator<AuditFinding> comparator =
        comparing(Chapter7::findingsImpact)
            .thenComparing(AuditFinding::name);
    return comparator.compare(a, b) > 0 ? y : x;           #A
}
```

#A Only taking the largest value

As usual, noticing one algebraic shape often reveals another. Each of our operations are just specializations of the built-in binary operator `maxBy`.

Listing 7.45 BinaryOperator's MaxBy

```
public static <T> BinaryOperator<T> maxBy(Comparator< T> comparator) { #A
    return (a, b) -> comparator.compare(a, b) >= 0 ? a : b; #B
}

static Policy add(Policy x, Policy y) {
    Comparator<Policy> comparator =
        comparing(Chapter7::policyImpact)
            .thenComparing(Policy::name);
    return maxBy(comparator).apply(x, y)      #C
}
static AuditFinding add(AuditFinding x, AuditFinding y) {
    Comparator<AuditFinding> comparator =
        comparing(Chapter7::findingsImpact)
            .thenComparing(AuditFinding::name);
    return maxBy(comparator).apply(x, y)      #C
}
```

#A MaxBy is built into Java's BinaryOperator interface

#B It turns comparators into Binary operations

#C Refactoring to use maxBy. This doesn't change any behavior, only make it explicit

This refactor comes with something interesting. If you stare at Listing 7.45, you might notice another unifying pattern born out of the fact that everything is built around binary operations. If we were so inclined, we could refactor again so that we don't even have to define any functions ourselves – we could use the binary operations Java already wrote for us!

Listing 7.46 Refactoring to static definitions

```
static BinaryOperator<Policy> addPolicies =          #A
    maxBy(comparing(Chapter07::policyImpact)
        .thenComparing(Policy::name));

static BinaryOperator<AuditFinding> addFindings =  #A
    maxBy(comparing(Chapter07::findingsImpact)
        .thenComparing(AuditFinding::name))

static BinaryOperator<Boolean> addPremiumStatus =  #B
    Boolean::logicalOr                         #B
```

#A Letting Java define the functions for us.

#B We can even do it with Boolean. Logical OR already exists, why should we redefine it?

Whether this refactoring is better or worse I leave to your sense of style, but it's pretty cool that we can do it.

7.8.1 The properties that define what it means to be correct

Let's finally write down a formal, rigorous specification of what it means to *correctly* combine conflicting data in our domain.

Listing 7.47 A formal specification of our requirements

```
a, b, c, d      ConflictingRows

Properties:
a + b = "a more favorable record" (whatever that means) #A
Monotonicity: a ≤ c AND b ≤ d THEN a + b ≤ c + d #B
Associativity: (a + (b + c)) = ((a + b) + c)
Ordered:
Reflexivity: a ≤ a
Antisymmetry: IF a ≤ b AND b ≤ a THEN a = b
Transitivity: IF a ≤ b AND b ≤ c THEN a ≤ c
```

#A Now we know what it means!

#B For an implementation to be correct, order must be preserved during merging

This small handful of properties is remarkably powerful. With them, we can write a test suite that will *prove* our implementation is correct -- because we know what correct means! We know every single state our code can enter, and what invariants must hold between them.

As usual, this process is several lengthy pages in a book, but just a few minutes of effort in real life (once you get used to it). It's mostly pattern recognition -- "oh, that's pretty close to [a property I recognize], I wonder if I could make it satisfy..." And more often than not, it's just brute force: "can I make this API be Associative? Is it ordered? Monotonic...?" If you can find one that fits, you reap all the rewards of thousands of years of refinement.

7.9 Jumping between universes

So far, we designed our entire algebra while ignoring Optional. Every binary operator assumes concrete values.

Listing 7.48 The design of our algebraic binary operators assumes data will be present

```
BinaryOperator<Policy> ...      #A
BinaryOperator<AuditFindings> ... #A
BinaryOperator<Boolean> ...      #A
```

#A No Optional values in sight!

But our data is sparse and messy and filled with non-values. Here's the actual model.

Listing 7.49 The messy reality that every field is potentially null

```
record RawData(
  String id,
  Optional<Policy> policy,          #A
  Optional<AuditFindings> findings, #A
  Optional<Boolean> premiumStatus   #A
){}
```

#A The messy reality is that every row is potentially a sea of undefined data

This omission wasn't a mistake. It's actually one of the coolest parts of this algebra stuff: we can *transform* an algebra just like we can transform data. We don't specifically

have to design *for* Optional data -- or even know it exists! We only need to have a way of converting a “normal” type into an Optional type.

My favorite way of thinking about these kinds of transformations is to imagine Optional as being its own parallel universe.

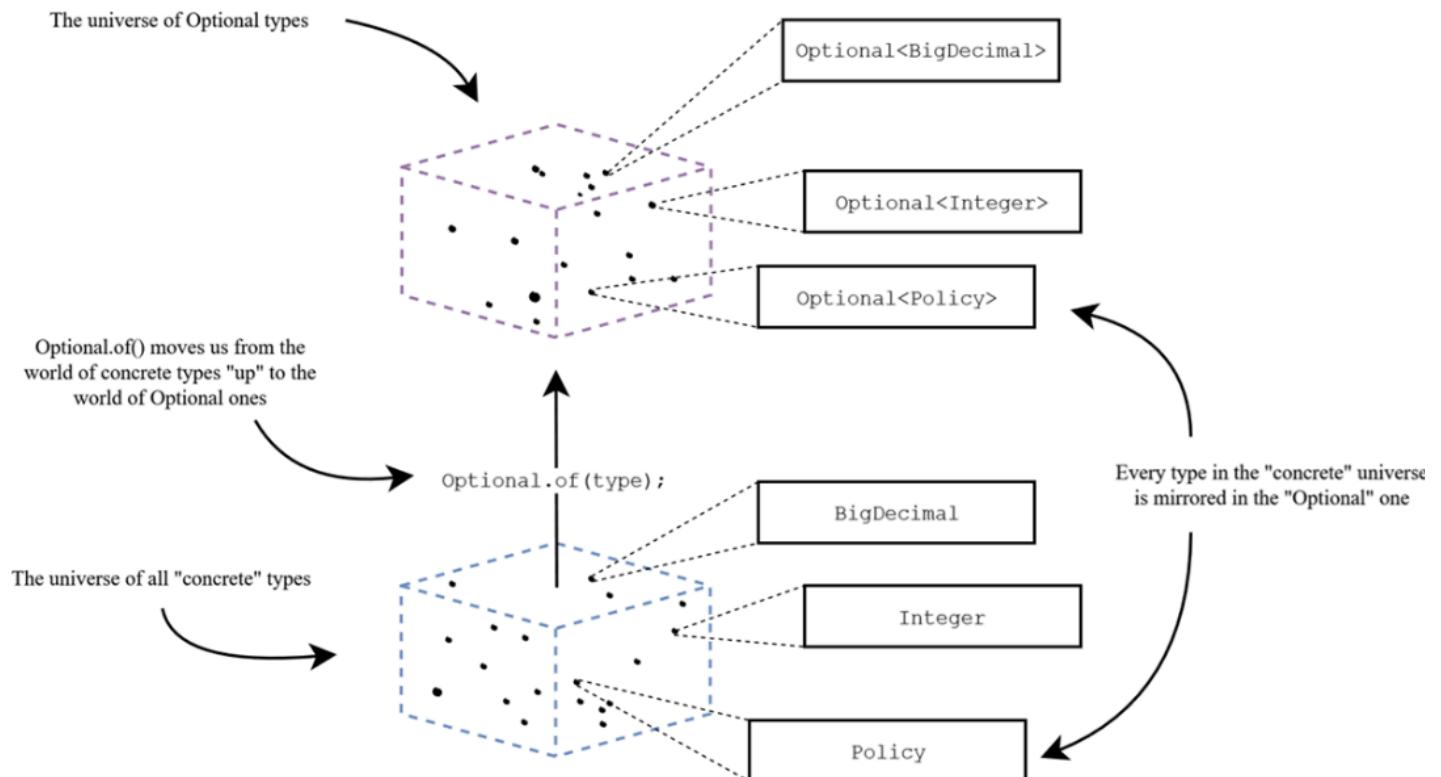


Figure 7.31 Visualizing Optional as a parallel universe

This parallel universe is a complete copy of the universe of “concrete” data types (like String, Integer, and AuditFinding), but where every type is wrapped in an Optional.

We can jump between these universes using special transformation functions. Any “normal” data type can be transported into the Optional one by calling `Optional.of`.

However, note that this jumping isn’t always bidirectional! It’s not always possible to get back “down” to the normal universe once you’ve turned your data into an Optional. `Optional.empty()` has no equivalent in the “normal” universe.

Here’s the powerful part: we can move *anything* between these two universes – even behavior itself! This technique is commonly called “lifting”.

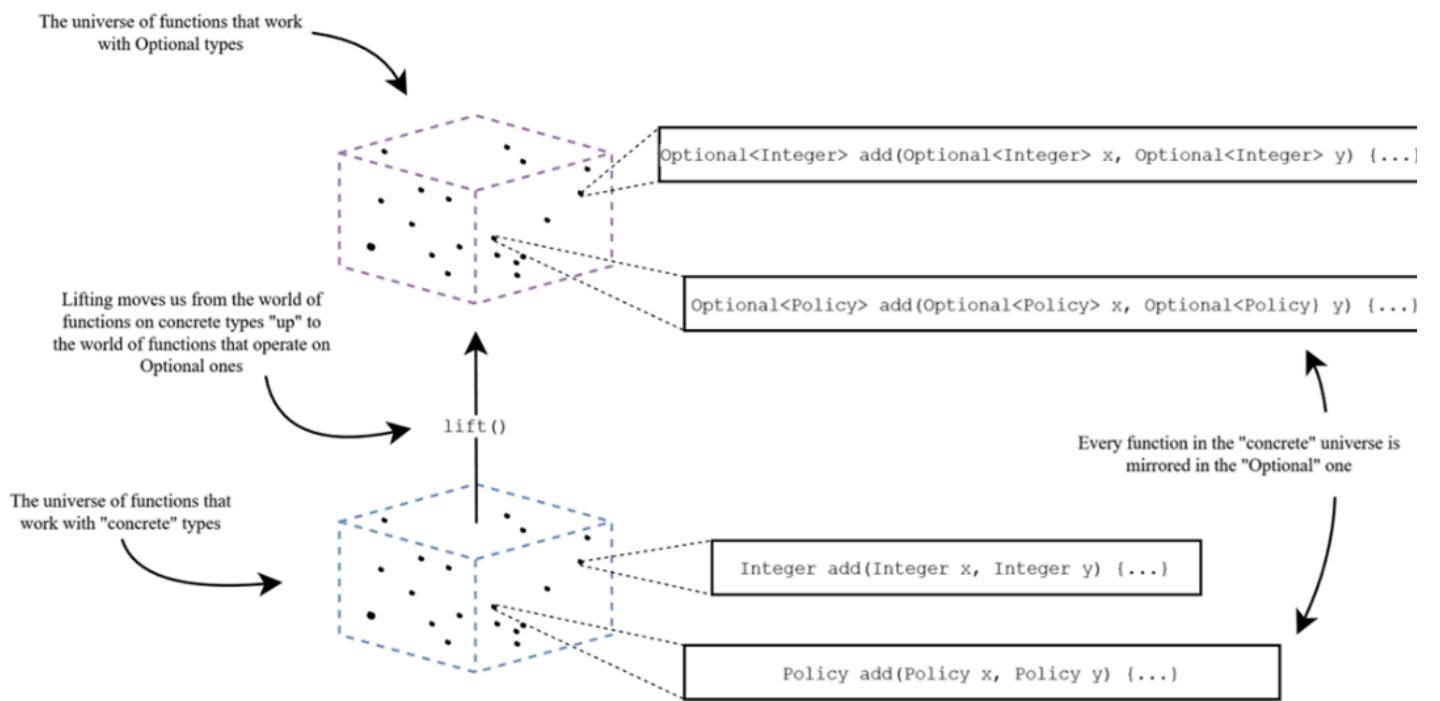


Figure 7.32 Jumping entire functions to the Optional universe

Lifting is what allows us to move our algebra into the world of Optional without writing a bunch of tedious mapping code by hand. However, before we see how to do it, we have to address something.

7.9.1 Dealing with the IDE telling us we're doing it wrong

We're going to write functions that take Optional as an argument. This is usually condemned during code reviews. Even IDEs give you scary warnings if you try.

Listing 7.50 Optionals as arguments?!

```
static <A> Optional<A> add(
    Optional<A> option1,    #A
    Optional<A> option2    #A
) {
    // ...
}
```

#A My IDE scolds me with "Optional is primarily intended for use as a method return type"

Ignore the IDE overlords and their conservative warnings! It's A-OK to build *tooling* that manipulates data types like Optional.

Here's a decent rule of thumb: as long as you have no idea what's inside of the Optional (i.e. it's holding a generic type variable), then taking it as an argument is fine.

Listing 7.51 This is OK because we never know what's inside of the Optional

```
static <A> Optional<A> doSomething(BinaryOperator<A> operator) {    #A
    return ...
}
```

#A We have no idea what's inside of the Optional. All we see is a generic type variable.

However, if you know what's inside of the Optional and start doing things with its concrete type, you're generally treading into something that's better solved with alternative approaches.

Listing 7.52 This knows too much about what's inside of the optional

```
static void doSomething(Optional<Integer> maybeInt) { #A
    if (maybeIn.isPresent()) {
        int myValue = maybeInt.get(); #B
    }
}
```

#A This usage would be suspect because we know what's in the Optional
#B The implementation ends up coupled to the details of what the optional is holding

But these are just rough guidelines. I'll break these "rules" all the time if I think it makes the code clearer. The important thing is that we don't avoid an entire pattern in programming just because IDEs or "best practices" tell us not to.

7.9.2 Lifting Binary Operator into the universe of Optional

Lifters take functions from the "normal" universe as input and return functions from the "Optional" universe as output. It's a generalization of this pattern:

Listing 7.53 The basic pattern lifters follow

```
static Optional<Integer> add(
    Optional<Integer> a,      #A
    Optional<Integer> b) {    #A
    if (a.isPresent() && b.isPresent()) { #B
        return Optional.of(a.get() + b.get()); #C
    } else {
        return Optional.empty(); #D
    }
}
```

#A This violates our rule of thumb about not knowing what's in the Optional for sake of example
#B First we check if both are defined
#C If so, we grab the values and perform some operation on them
#D Otherwise, we return Optional.empty()

We can derive generic lift functions in the same way we derived a generic toMap back in section 7.3: extract the common patterns.

The first thing we could pull out from listing 7.53 is the hard coded binary operation that's adding the two integers. We could pass that in and, in doing so, turn this into a function that can "translate" any binary operator from the normal world into one that knows how to operate in the Optional one.

Listing 7.54 Refactoring to pass in the BinaryOperator

```
static Optional<Integer> apply(          #A
    BinaryOperator<Integer> operator,      #B
    Optional<Integer> a,
    Optional<Integer> b) {
    if (a.isPresent() && b.isPresent()) {
        return Optional.of(operator.apply(a.get(), b.get())); #B
    } else {
        return Optional.empty()
    }
}
```

#A The function is no longer about just adding two Optional ints, we we change it to a more generic “apply”

#B Taking the binary operator as an argument lets us evaluate it inside of the world of Optional

Once you make it here, you’d probably question why we bother specifying Integer at all. We could make this about any concrete type from the normal world.

Listing 7.55 Refactoring to generic types

```
static <A> Optional<A> apply(          #A
    BinaryOperator<A> operator,      #A
    Optional<A> a,                  #A
    Optional<A> b) {              #A
    if (a.isPresent() && b.isPresent()) {
        return Optional.of(operator.apply(a.get(), b.get()));
    } else {
        return Optional.empty()
    }
}
```

#A Making our apply method work with any type from the normal universe

To make this a lifter, we just have to perform one more leap of abstraction and decouple the BinaryOperator from its arguments. We want to lift the BinaryOperator *itself* into the Optional universe – not the arguments we pass to the Binary Operation. Those can be supplied later. It’s by breaking these two things apart that we arrive at a general purpose function that can lift binary operations between worlds.

Listing 7.56 An implementation of lift

```
static <A> BinaryOperator<Optional<A>> lift(BinaryOperator<A> f) { #A
    return (opt1, opt2) -> { #B
        if (opt1.isPresent() && opt2.isPresent()) { #C
            return Optional.of(f.apply(opt1.get(), opt2.get())); #C
        } else { #C
            return Optional.empty(); #C
        }
    }
}

static <A> BinaryOperator<Optional<A>> lift(BinaryOperator<A> f) { #D
    return (opt1, opt2) -> opt1.flatMap(
        x -> opt2.map(
            y -> f.apply(x, y)));
}
```

#A Check out our type signature. We take a BinaryOperator from the normal universe and return a BinaryOperator in the Optional one

#B This is where we “moved” the arguments. They’re now returns as part of the new BinaryOperator we’re creating with a lambda

#C The rest of the implementation is business as usual

#D If you prefer a more functional style, here’s the same thing using map and flatMap

Now we can lift any `BinaryOperator` in the normal universe into one that works in the Optional one!

Listing 7.57 Converting BinaryOperators between worlds

```
BinaryOperator<Optional<Integer>> maybeAdd = lift(Integer::sum); #A
BinaryOperator<Optional<String>> maybeConcat = lift(String::concat); #A
```

#A Lifting arbitrary binaryOperations into the Optional universe

Lifted functions get rid of the need to manually unpack, presence check, or map Optional data by hand. That plumbing is built into the lifted function! It allows us to program with Optional data in a way that feels like passing around normal values.

Listing 7.58 using lifted functions with Optional data

```
assertEquals(
    maybeAdd.apply(Option.of(1), Option.of(2)), #A
    Option.of(3) #A
);
assertEquals(
    maybeConcat(Option.of("Hello"), Option.of("World")), #A
    Option.of("HelloWorld") #A
)
assertEquals(
    maybeConcat(Optional.empty(), Option.of("World")), #B
    Optional.empty() #B
)
```

#A The lifted functions automatically apply their operation when all of the optionals are present

#B Otherwise, they return an empty value

How crazy is that? When I first learned about lifting my mind was blown. I couldn’t believe that entire functions could be moved between universes like this. It frees us to

spend most of our design effort in the happy world where everything is defined and then just sprinkle on capabilities like Optional handling later.

7.9.3 Moving our algebra into the universe of Optional

Now that we can lift arbitrary binary operators, we can use them to transform our algebra into one which works with Optional values.

Listing 7.59 Lifting our Algebra into the Optional universe

```
static <A> BinaryOperator<A> withOptional(BinaryOperator<A> operator) {
    return (opt1, opt2) ->
        lift(operator).apply(opt1, opt2)                      #A
            .or(() -> opt1.isPresent() ? opt1 : opt1);      #B
}

static BinaryOperator<Optional<Policy>> addPolicies =           #C
    withOptional(maxBy(comparing(Finalizing::policyImpact)
        .thenComparing(Policy::name)));

static BinaryOperator<Optional<AuditFinding>> addFindings =   #C
    withOptional(maxBy(comparing(Finalizing::findingsImpact)
        .thenComparing(AuditFinding::name)));

static BinaryOperator<Optional<Boolean>> addStatuses =         #C
    withOptional(maxBy(Boolean::compareTo));
```

#A First we lift up to the world of Optional. This will choose a winning value if they're both defined.

#B If they're not both present, maxBy will take whichever is more favorable

#C Transforming our algebra into one that's Optional aware

And that's all it takes to handle optionality in our domain!

But is it *correct*? Does this really satisfy all of our properties?

I leave that to you as an exercise! All of the properties are in this chapter. You just have to extend them to the new data type.

7.10 The finished implementation

Here's the finished implementation.

Listing 7.60 Here's the whole thing

```
static BinaryOperator<Optional<Policy>> addPolicies =
    withOptional(maxBy(comparing(Finalizing::policyImpact)
        .thenComparing(Policy::name)));

static BinaryOperator<Optional<AuditFinding>> addFindings =
    withOptional(maxBy(comparing(Finalizing::findingsImpact)
        .thenComparing(AuditFinding::name)));

static BinaryOperator<Optional<Boolean>> addStatuses =
    withOptional(Boolean::logicalOR);

static RawData merge(RawData x, RawData y) {
    if (!x.id().equals(y.id())) {
        throw new IllegalArgumentException(...)
    }
    return new RawData(
        x.id(),
        addPolicies(x.policy(), y.policy()),
        addFindings(x.findings(), y.findings()),
        addStatuses(x.premium(), y.premium())
    );
}

static List<RawData> cleanDuplicates(Collection<RawData> rows) {
    return List.of(toMap(rows, (rowA, rowB) -> add(rowA, rowB).values()));
}
```

That's the whole thing. Just a handful of lines of code. This is the power of algebraic thinking. We can do more with less. This is what true software reuse looks like.

7.11 A final note on representing algebraic ideas in code

We've spent this book trying to figure out how to get things we know out of our heads and into the code. Now we know even *more* stuff! Obviously, we should put it into the code, right? Well...

Listing 7.61 Should we make these?

```
interface Associative<A> extends BinaryOperator<A> {}
interface Reflexive<A> extends BinaryOperator<A> {}
interface Monotonic<A> extends BinaryOperator<A> {}
// and so on...
```

I'm going to argue that we don't.

The first reason is social. Engineering large systems is a group effort. That means concessions. Many developers hold a deep revulsion to anything "math-y". (This chapter will surely be the most contentious because of it.) Introducing them into a codebase can unfortunately cause a lot of grumbling and accusations of "trying to turn Java into Haskell." Avoiding these conversations leads to less overall friction on a project.

The second reason is that Java won't know anything about the special interfaces we create. The more you specialize, the more you'll isolate yourself from the existing

ecosystem. Often, the end result of this specialization is a set of half-implemented wrappers around the Java Collection classes and a bunch of casting between “normal” Java and your bespoke flavors.

So, that’s why we built everything on top of the regular Java standard library. It’s safe. It’s familiar. Best of all, it’s already there.

7.12 Wrapping up

The properties we covered in this chapter are what I consider the working horses of the algebraic world. They’re simple, pervasive, and powerful. They show up over and over again in daily programming. Learning to recognize them will transform your programming.

Next up, we’re going to build on these new algebraic powers to see how we can represent complex business rules as plain data that we can interpret in order to get an answer.

7.13 Summary

- Algebraic thinking enables us to rigorously define correctness
- If we know what correct means our tests suite can verify it
- Being able to describe correctness is important in the age of large Language Models
- Algebra doesn’t need to be fancy! An entire “algebra” can be a single operation
- We do algebra in Java by thinking about what equations hold between data types
- Algebraic properties are based around assertions that begin with “for all states...”
- Math uses an upside-down capital “A” (\forall) as a shorthand meaning “for all”
- Notation allows us express complex relationships between variables compactly ($\forall a, b, c$)
- Expressing ideas in mathematical notation forces us to be precise about what we mean
- Associativity is the most common property you’ll find in programming
- Associative operations can be grouped in any order ($\forall a, b, c \in X, (a + (b +)) = ((a + b) + c)$)
- Associativity enables parallel computation by decoupling us from sequential application
- Algebraic thinking can guide us towards simpler, smaller implementations
- Don’t force a property to work. Clear data semantics outweighs elegant algebras
- Commutativity means that “the order of arguments doesn’t matter”
- The Commutativity Law: $\forall a, b \in X, a + b = b + a$
- The ordering properties used by Comparator are
 - Reflexivity: $\forall a \in X, a \leq a$

- Antisymmetry: $\forall a, b \in X, \text{IF } a \leq b \text{ AND } b \leq a \text{ THEN } a = b$
- Transitivity: $\forall a, b, c \in X, \text{IF } a \leq b \text{ AND } b \leq c \text{ THEN } a \leq c$
- Orders are so “obvious” that they’re the easiest to overlook.
- Many problems can be solved by realizing they’re ordered (even minimally)
- Monotonicity describes functions which preserve the ordering properties of their data
- The monotonicity law: $\forall a_1, a_2, b_1, b_2 \in X, \text{IF } a_1 \leq b_1 \text{ AND } a_2 \leq b_2 \text{ THEN } a_1 + a_2 \leq b_1 + b_2$
- The hardest part of monotonicity is realizing that it’s not saying something complicated
- *Not adding algebraic interfaces into the code is usually a good call (but it’s up to you!)*
- BinaryOperator and UnaryOperator are useful Algebraic interfaces built into Java
- Algebraic properties can guide our implementation in both the small and large
- Designing around an algebra lets you leverage Java’s most powerful tooling
- An implementation doesn’t need to match the algebra!
- With practice, you can see the algebra that underpins any implementation
- Studying algebraic properties lets us see the commonalities that unite programming
- A useful mental model for Optional is to think of it as being its own parallel universe
- We can move data, functions, and behavior into the Optional universe using transforms
- Optional.of moves data from the “normal” universe into the Optional one
- A special function called “lift” can move functions into the Optional universe
- Algebraic thinking has the same effect as data modeling. It simplifies everything.

8 Business Rules as Data

This chapter covers

- Modeling business rules as data
- Interpreters and DSLs
- Not over thinking it and just writing code

Java lives in the shadow of the language it's perceived to be. It gets unfairly associated with patterns that were in vogue at the time when it rose to prominence. We constructed cathedrals of inheritance, bizarre abstractions, and there wasn't a problem that we couldn't "solve" with a design pattern.

But the language is not the patterns. It never had to be that way.

Java has continued to evolve, but many developers are still living in its former shadow. The perceptions around how we should build, and what it costs to do so, linger in the community's unconsciousness.

Modern Java is lean and confident. Records and pattern matching have fundamentally changed how we approach design in Java. New patterns are available to us and leveraging them starts with data.

8.1 A few requirements

At MegaCorp, large customer accounts have dedicated sales teams. These teams help customers navigate complex contracts, issue special discounts, and so on. Their commission is tied to the type of accounts they manage and the value of the deals they cut, so which account ends up under which team is steeped in historical decision, shifting org charts, and petty in-fighting. There're no clean rules like "everything in the USA is handled by the US based sales team." If only! Instead, it's based on arbitrary and ever-changing rules.

We have rules like this: If the customer is in AMER region and public-sector then it goes to Bob's team. Private sector? That's Jane's territory. All of Europe? Joe. Except for Enterprise accounts in Belgium. Those belong to Gregory. And so on.

Customer Accounts have a large set of geographic and strategic sales attributes

Each sales org carves out the set of customers in which they're interested in handling.

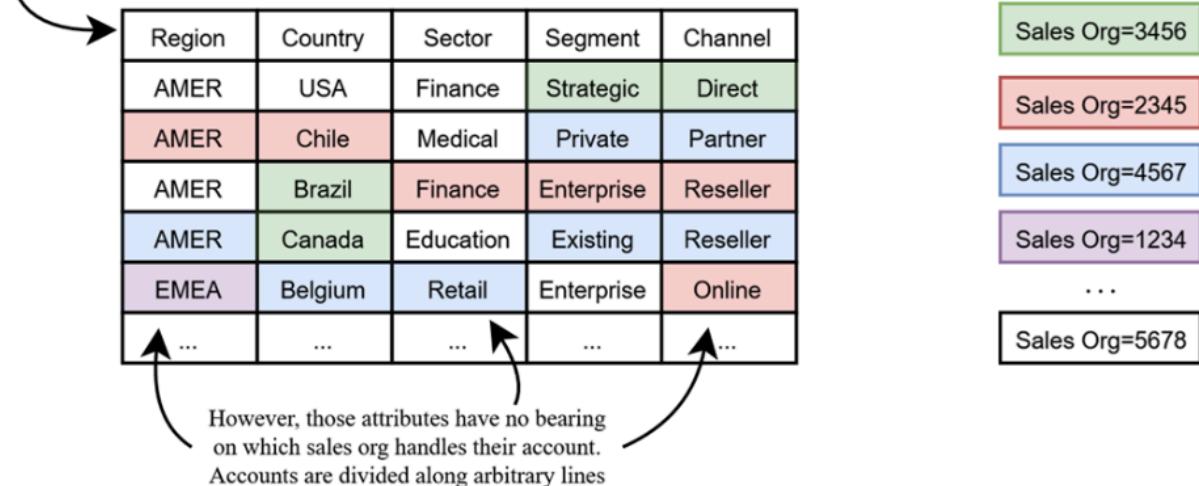


Figure 8.1 The challenge of routing a customer account to the right sales organization

Individually, none of the rules are complex. The challenge comes from their scale and intermingling. There are dozens of different rules for how accounts should be divided. They hand them to us in as a big text file like this:

We get a big text file filled with the rules they want applied

Rule #1
All accounts in EMEA excluding those in Belgium, Austria, and France belong to SalesOrg=111

Rule #2
All non-reseller accounts in Belgium, Austria, and France belong to SalesOrg=222

Rule #3
Resellers in Belgium belong to SalesOrg=333

Rule #4
Resellers in Austria and France belong to SalesOrg=444

...

(and so on for 50 more rules)

Each of these needs translated into code

Figure 8.2 What we're working with

We've also got deadlines to hit, so there are competing pressures. We have to get these into the code now, but also make sure we don't code ourselves into a corner. We don't want to get stuck managing these rules by hand forever. Our design should be flexible enough that we can turn it into a tool for the business people to use on their own.

In a real-life system, an Account is probably made up of dozens (if not hundreds) of individual fields fractured across various databases, data lakes, and services. Our

system surely has its own Entity made up of a hodgepodge of attributes. And we surely have to interact with a few external services to build a complete view of an Account.

But that's gross.

Instead, we'll do the data-oriented thing and just build the world we want out of plain data. We don't need The Official Account Entity with all of its baggage; we just need enough of a data model to get our job done. We can use the modeling technique we explored in Chapter 5 to hide complexity we don't care about.

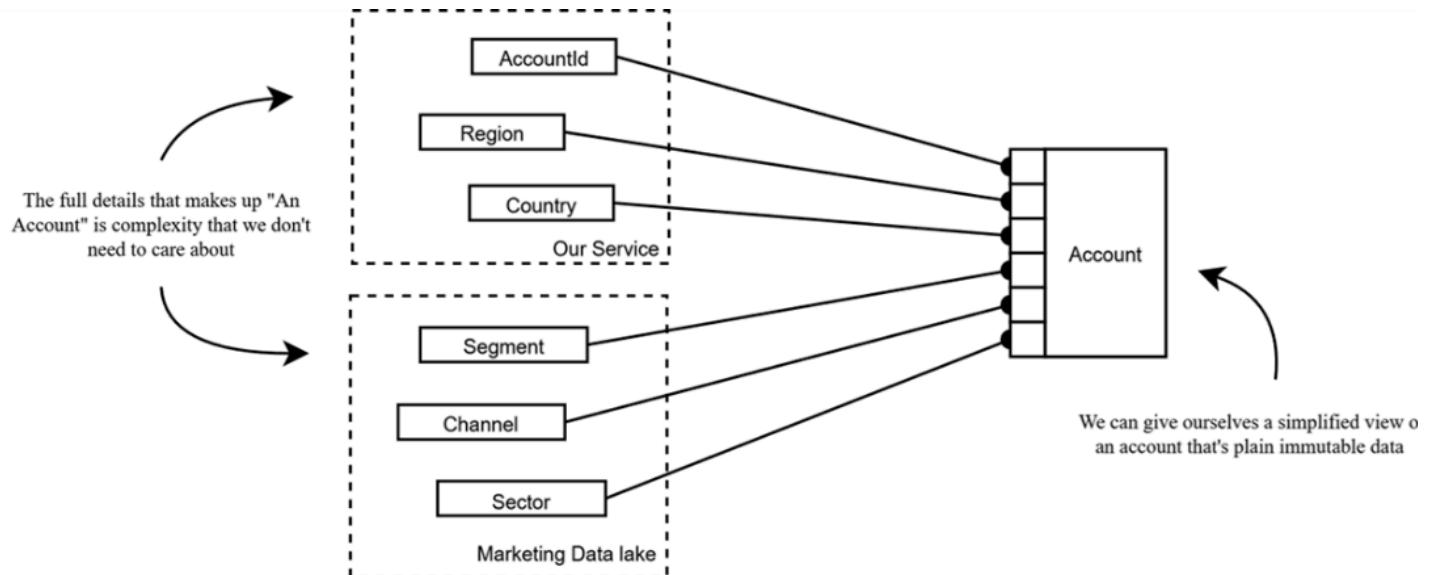


Figure 8.3 Good modeling hides complexity we don't care about.

In code, it might look something like this:

Listing 8.1 The customer account

```
record Account( #A
    AccountId accountId,
    Region region,
    CountryCode country,
    Sector sector,
    Segment segment,
    SalesChannel channel
){}
```

#A This information is what we use to choose a Sale Organization

The account is made up of multiple domain types that are a mix of enums and arbitrary strings that we've wrapped with custom records.

Listing 8.2 Defining what all our data means

```
enum Segment {Enterprise, Strategic, Existing, ...} #A
enum SalesChannel {Direct, Partner, Reseller, ...} #A
enum Region { LATAM, NA, EMEA, ...} #A
enum CountryCode {AC, AD, AE, ...} #A
record Sector(String value) {} #A
```

#A The individual values here don't matter beyond giving us something to write our rules against

That's the setup. But before we hop into code, *of course* we're going to ask...

8.1.1 What does it mean to be correct?

Even though we're modeling arbitrary, human invented, and ever-changing rules, we can still be formal about what it means for those rules to be correct. The humans writing these rules will only know about their slice of the world. Are the rules they give us valid? Do they break other rules? Should rules be allowed to overlap?

Here's where we'll cheat a little bit for the sake of simplifying things for the book. The real world is big and complex – rules will conflict, overlap, duplicate each other, and collide in just about every possible way. But we'll create a smaller, focused world where those problems aren't allowed to exist.

If we assume that a Rule is a function from Account to SalesOrgId like this

Listing 8.3 Assuming our rule function is modeled something like this

```
interface Rule extends Function<Account, Optional<SalesOrgId>> {} #A  
  
public static Optional<SalesOrgId> orgId1234rule(Account account) { #B  
    return ...  
}
```

#A A "Rule" in our system is a deterministic function from Account to SalesOrgId (if one exists)
#B Meaning our functions will look something like this

Then we can enforce some simplifying assumptions. For instance, no two rules should point at the same inputs and the same output. Similarly, no two rules should point the *same* inputs at *different* outputs. We'd have no way of knowing which output was the right one.

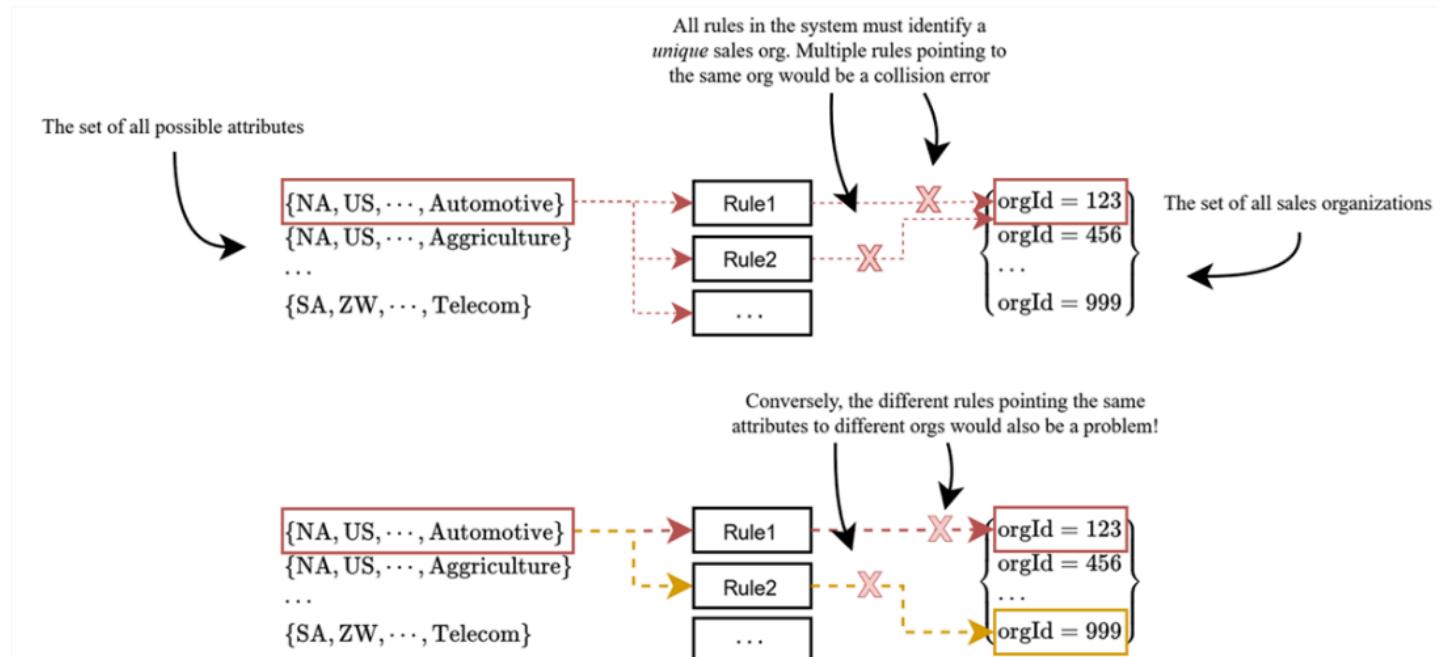


Figure 8.4 What obviously shouldn't happen. How would we pick which rule was right?

We can't say anything about the correctness of an individual rule – that's up to the human writing it, but we can say what it means for the behavior of all the rules in aggregate to be correct.

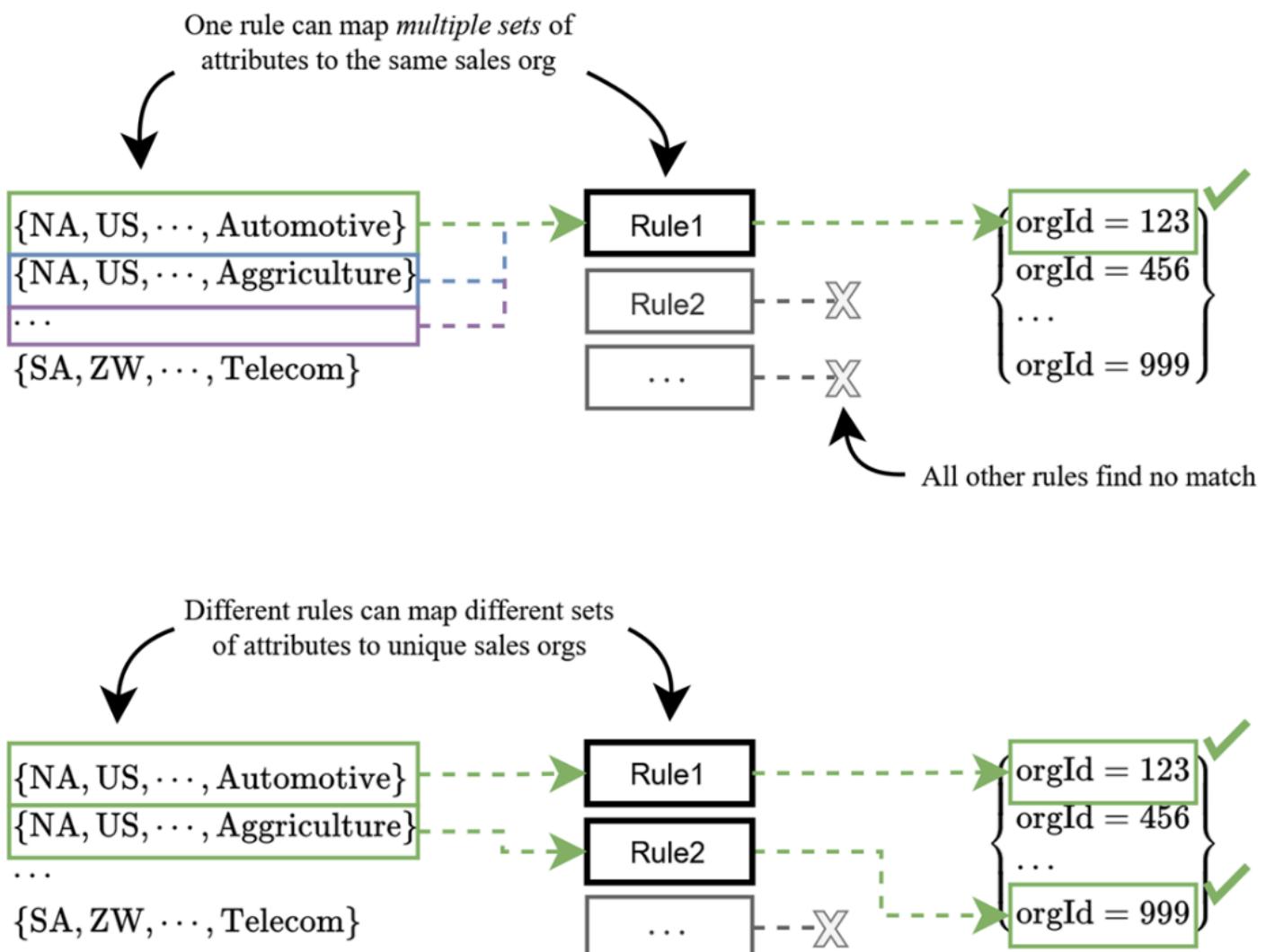


Figure 8.5 Rules must each map to a unique org.

From this, we can come up with a simple global property. For all rules in the system, and for all possible inputs, it must hold that if two rules map the same inputs to the same output, then they're the same rule.

Listing 8.4 Imagining something like...

```
List<Rule> allKnownRules = List.of(...);    #A

@Test
void noRulesOverlap() {
    for (Account account: allPossibleAccounts()) {          #B
        for (Rule a : allKnownRules) {
            for (Rule b : allKnownRules) {
                if (a.apply(account).equals(b.apply(account))) { #C
                    assertEquals(a, b);                            #C
                }
            }
        }
    }
}
```

#A Collecting all the rules in our program into a list

#B Assume that we've defined this elsewhere (like we did in the last chapter)

#C The main assertion: if two rules send the same input to the same output, then they **MUST** be the same rule. Otherwise, those rules overlap and it's an error. (Note that we can't actually compare functions like this, it's here just for example)

With this defined, we can have our test suite keep an eye on us as we work through writing the rules. If we find one that overlaps, or mess up an input, our test suite will catch it.

8.2 Muscling through the rule implementation

We might start by translating the rules directly into code like this.

Listing 8.5 Our first rule! Only dozens more to go after this one...

```
/**
 * Rule #1
 * All accounts in EMEA excluding those in Belgium, Austria,
 * and France belong to SalesOrg=111
 */
static Optional<SalesOrgId> rule0rg1234(Account account) {
    if (account.region().equals(Region.EMEA)) {
        Set<CountryCode> excluded = Set.of(BE, AU, FR);
        if (!excluded.contains(account.country())) {
            return Optional.of(new SalesOrgId("111"));
        } else {
            return Optional.empty();
        }
    } else {
        return Optional.empty();
    }
}
```

But those if/else statements are noisy. Most devs would refactor to simplify down to a single Boolean expression like this:

Listing 8.6 Ah... much better?

```
/**  
 * Rule #1  
 * All accounts in EMEA excluding those in Belgium, Austria,  
 * and France belong to SalesOrg=111  
 */  
static Optional<SalesOrgId> ruleForOrg111(Account account) {  
    Set<CountryCode> excluded = Set.of(BE, AU, FR);  
    return account.region().equals(Region.EMEA)  
        && !excluded.contains(account.country())  
    ? Optional.of(new SalesOrgId("111"))  
    : Optional.empty();  
}
```

That feels like an OK pattern. Let's write another.

Listing 8.7 Writing another rule

```
/**  
 * Rule #2  
 * All non-reseller accounts in Belgium, Austria, and France  
 * belong to SalesOrg=222  
 */  
static Optional<BillingCode> ruleForOrg222(Account account) {  
    Set<CountryCode> included = Set.of(BE, AU, FR);  
    return account.region().equals(Region.EMEA)  
        && included.contains(account.country())  
        && account.channel().equals(SalesChannel.Reseller)  
    ? Optional.of(new SalesOrgId("222"))  
    : Optional.empty();  
}
```

Each rule is simple, but we have a lot of them to write, and each one requires lots of code. A larger problem of scale is emerging. Debugging one or two of these rules is easy enough. Debugging multiple will be a nightmare. The initial refactor away from nested if/else statements improved how the code looks, but destroyed our ability to debug and monitor it. Everything is compressed down to a single yes / no. There's nowhere to drop a break point to see why our code is making a particular decision.

This problem will grow into an operational nightmare in production. Even if we log out *which* rule matched, we'll have no idea *why* it did so. The Booleans are destroying information. Simple questions like "Why did(n't) you guys send account X to Sales Org Y?" will be impossible to answer without digging into the code and manually unwinding each condition so we can evaluate it to see what did or didn't match.

There's a world where we solve this by going back to if/else statements and dumping logs everywhere.

Listing 8.8 Operational insight, but terrible code

```
/**  
 * Rule #1  
 * All accounts in EMEA excluding those in Belgium, Austria,  
 * and France belong to SalesOrg=111  
 */  
static Optional<SalesOrgId> ruleOrg1234(Account account) {  
    if (account.region().equals(Region.EMEA)) {  
        log.info("matched region=EMEA");  
        Set<CountryCode> excluded = Set.of(BE, AU, FR);  
        if (!excluded.contains(account.country())) {  
            log.info("matched: {} not in {}", account.country(), excluded);  
            return Optional.of(new SalesOrgId("111"));  
        } else {  
            log.info("Expected country not in {}, found country={}",  
                excluded,  
                account.country())  
            return Optional.empty();  
        }  
    } else {  
        log.info("expected region=EMEA, found region={}",  
            account.region());  
        return Optional.empty();  
    }  
}
```

This has the benefit of giving useful runtime information, but the odds of any developer doing this, or being able to maintain it over time, are effectively zero. There's also the problem that this approach fails our long-term goal of being able to offload the creation of these rules to the business people. Everything is hard coded; nothing is portable.

It's about this point where most developers will start looking for a way out. However, the options aren't that great. Popular Rules Engines and DSLs are behemoths. They often require custom grammars or string based "Expression Languages" that leave us cramming rules into Annotations. A modern option might be to offload these to your favorite LLM, but that only makes generating the bad code faster. You're still on the hook for debugging the output and getting paged when its wrong.

So, instead of pulling in a dependency, we're going to build our own "rules engine" directly in plain, vanilla, dependency-free Java. This is something that would have been pretty expensive in the past, but records and pattern matching have fundamentally changed what certain patterns cost. Modern Java allows us to create powerful abstractions with very little code.

All we have to do is start with data.

8.3 Capturing rules as plain data

Throughout this book, we've learned to see data in new ways. It started with the familiar. Things like `Degrees` and `NonNegativeInts`. Then we hopped up a level of abstraction. We used data to represent "things we know," like an invoice being `PastDue`, or `NonBillable`. Then we jumped up yet another level of abstraction and represented *decisions* in the code as plain data. We did this back in chapter 5.

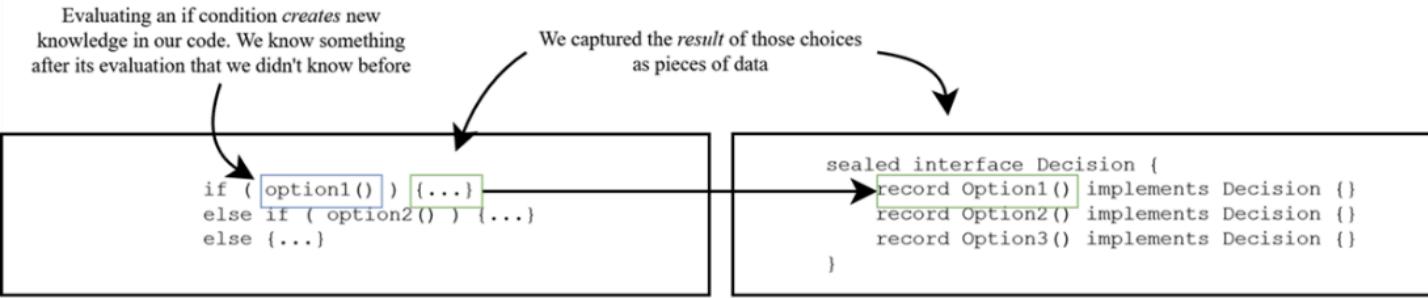


Figure 8.6 Capturing what we know after evaluating a condition as data

Now we're going to take one more leap of abstraction and represent the code that *leads* to those decisions as data. Operations themselves can be data!

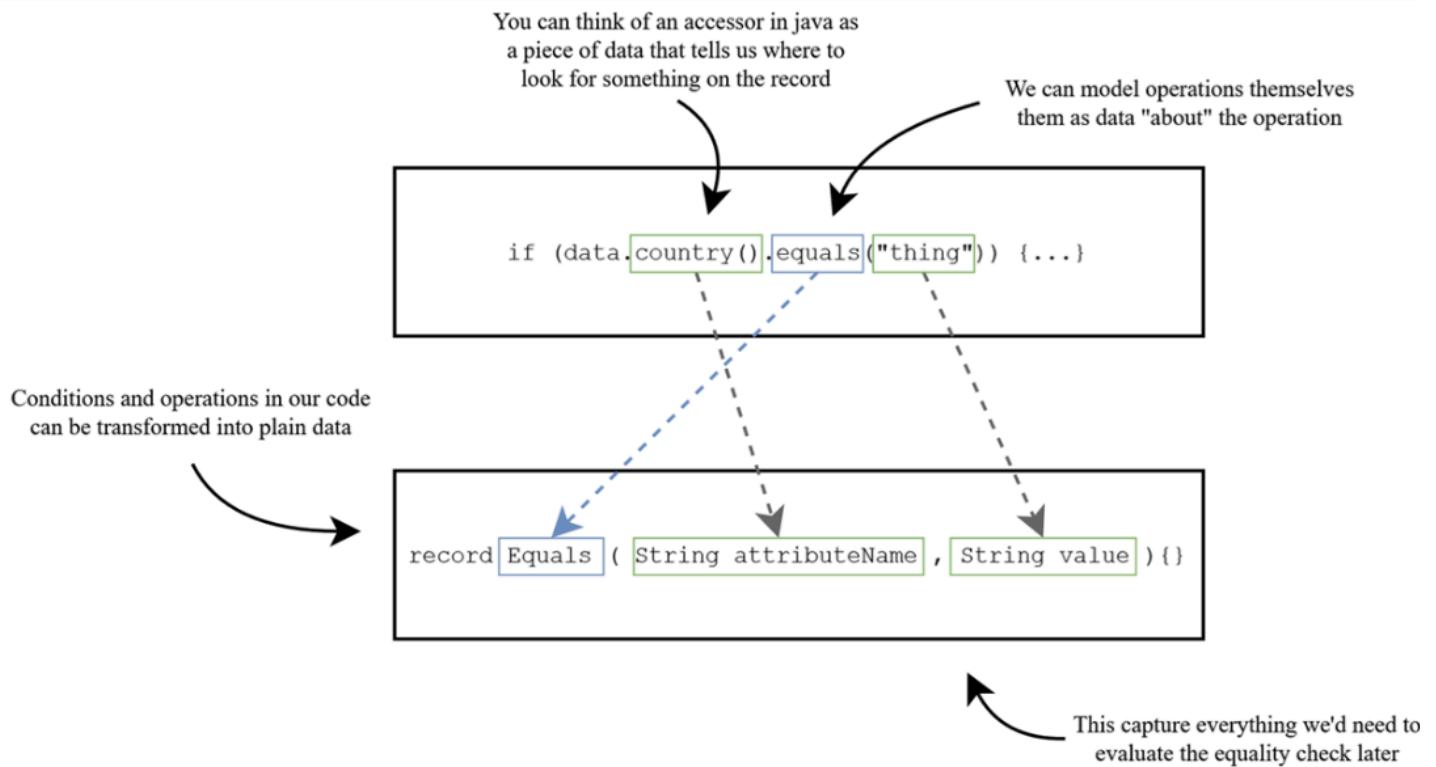


Figure 8.7 Modeling operations as data

This is data about an equality check we want to evaluate *later*.

Listing 8.9 Expressing rules as plain data

```
Equals rule1 = new Equals("country", "US"); #A
Equals rule2 = new Equals("country", "BE"); #A
Equals rule3 = new Equals("country", "FR"); #A
```

#A All of these describe an equality check that we want to perform later

On its own it's not that impressive, and it has some obvious shortcomings that we'll address later, but turning operations into data is the first step towards something extremely powerful. Now that we have data, we can build an algebra!

But first, let's fix something on this `Equals` data type. It has too many strings. They allow us to pass invalid attribute names which don't exist on the `Account` data type.

Listing 8.10 Typos will be our toughest enemy

```
record Equals(String attributeName, String value); #A
Equals rule1 = new Equals("county", "US"); #B
Equals rule2 = new Equals("i-am-not-valid", "BE"); #C
```

#A This String allows many crimes to be committed
#B Good luck finding this typo in a sea in a sea of similar rules
#C We can supply any garbage we want here

Instead of a plain string, for now, we'll model all of the attributes on the Account type as an Enum. This gets us out of the land where typos will be our biggest headache.

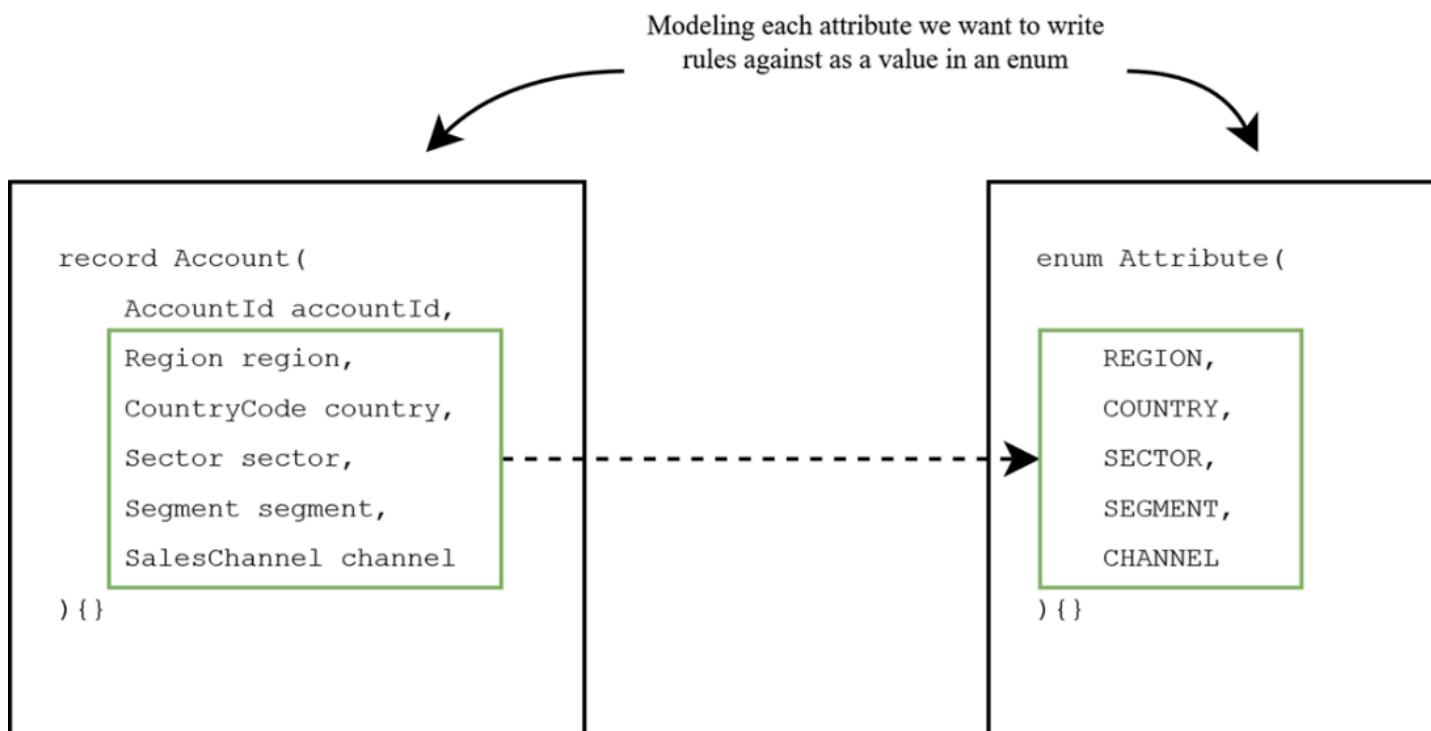


Figure 8.8 Creating an enum that models the relevant attributes

That lets us refactor the Equals type to lock down the attribute to only things which exist.

Listing 8.11 Refactoring to use an enum instead of wide-open String

```
enum Attribute {REGION, COUNTRY, SECTOR, SEGMENT, CHANNEL}; #A
record Equals(String attributeName, String value){} #B
record Equals(Attribute attribute, String value) {} #B
```

#A Modeling all of the attributes as an enum
#B Swapping out String for Attribute

We'll eventually fix that other String, too. But this is good enough for now. The nice part about modeling with algebraic data is that we don't have to worry about getting it 100% right on the first try. We can tune it as we go.

8.4 Building a simple Algebra

Rules are more than just simple equality checks. They're filled with compound conditions like ANDs and ORs. We can model these in exactly the same way: turning the operations into data.

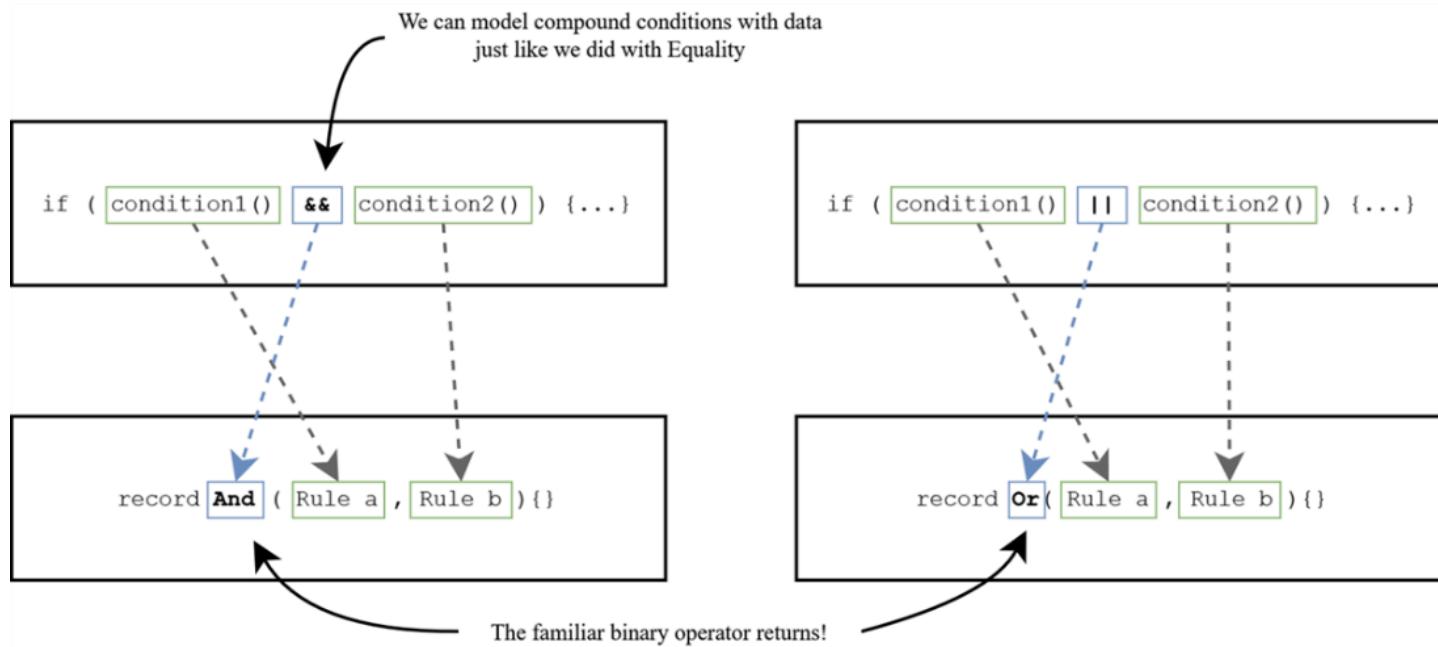


Figure 8.9 Modeling logical AND and OR as plain data

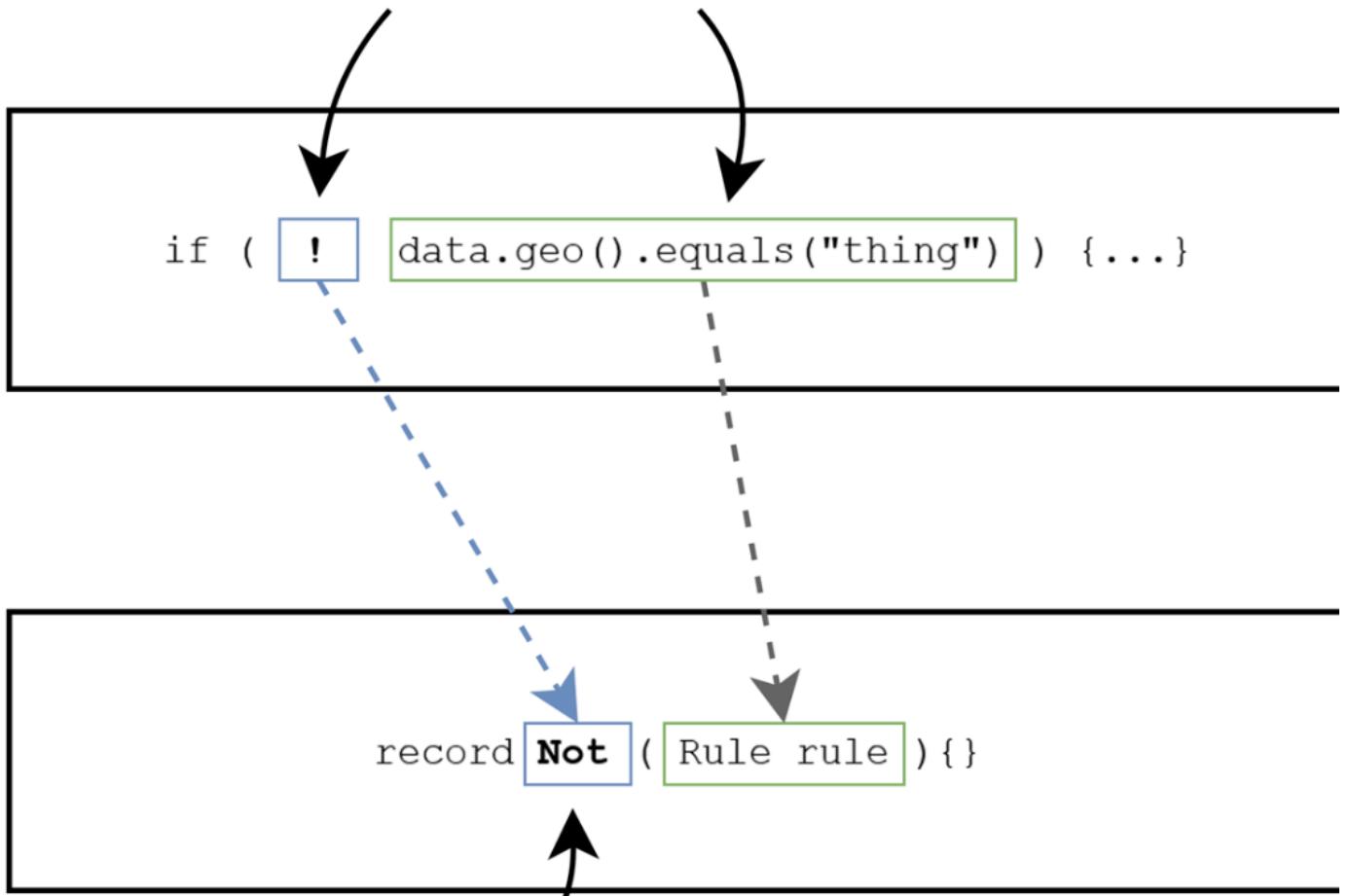
These new data types can form a closed algebra by unifying them under a single sealed interface (sum type).

Listing 8.12 An algebra appears!

```
sealed interface Rule {  
    record Equals(Attribute field, String value) implements Rule {}  
    record And(Rule a, Rule b) implements Rule {}  
    record Or(Rule a, Rule b) implements Rule {}  
}
```

These 3 data types are enough to express any inclusive style rule, but we also need to express rules that *exclude* values (e.g. "Region should *not* equal EMEA"). We need to model an operation that negates values in our algebra.

The `!` operator in Java negates everything that comes after it



We can express the same negation as plain data.

Figure 8.10 Modeling negation as plain data

That was easy enough. Here's the whole thing.

Listing 8.13 The finished algebra

```
sealed interface Rule {  
    record Equals(Attribute field, String value) implements Rule {}  
    record And(Rule a, Rule b) implements Rule {}  
    record Or(Rule a, Rule b) implements Rule {}  
    record Not(Rule rule) implements Rule {}  
}
```

This tiny algebra is actually enough to express all of the business rules we need for our requirements (and more!). Algebraic approaches allow small things to build much bigger ones.

Although, “small” might not be the best word to describe what we’ve currently built. It’s pretty clunky and verbose.

Listing 8.14 It's super verbose right now, but we'll make it better

```
new Or(
    new And(new Equals(COUNTRY, "US"), new Equals(SEGMENT, "Public")), #A
    new Not(new Equals(REGION, "AMER"))); #A
```

#A Creating rules currently involves a lot of boilerplate

But what's neat is how quickly this becomes an absolute delight to use with just a few tweaks. First, we can get rid of the `new` boilerplate by creating a few static constructor methods.

Listing 8.15 A few convenience static constructors

```
static Equals eq(Attribute attr, String val) { #A
    return new Equals(attr, val);
}
static Not not(Rule rule) { #A
    return new Not(rule);
}
```

#A These just wrap up the constructors and let us call them without the new ceremony

Next we can add default methods on the `Rule` interface. These enable fluent chaining of `and` / `or`.

Listing 8.16 Adding default methods to support chaining

```
sealed interface Rule {
    // ...
    default Rule and(Rule other) {
        return new And(this, other);
    }
    default Rule or(Rule other) {
        return new Or(this, other);
    }
}
```

And suddenly we've got something really awesome.

Listing 8.17 Check this out!

```
if (account.region().equals(EMEA) #A
    && account.segment().equals(Public)) { #A
    ...
} #A
else if (!account.region().equals(LATAM)) { #A
    ...
} else { #A
    // no match #A
}

Rule rule = (eq(REGION, "EMEA").and(eq(SEGMENT, "Public"))). #B
    .or(not(eq(REGION, "LATAM"))); #B
```

#A Compound, multi-branch if/else statements in code

#B Can be expressed extremely tersely in our algebra

But what about more complicated rules like this one?

Listing 8.18 Can data handle this?

```
Set<CountryCode> allowed = Set.of("US", "FR", "BU");
if (allowed.includes(account.country())) { #A
    ...
}
```

#A Can we use data to express complicated ideas like “one of”?

Of course! It's just data! We can transform it however we want.

8.4.1 Building better interfaces into the algebra

This is where our algebraic skills from the previous chapter really start to shine. Compound actions, like containment or set inclusion, can be viewed as the repeated application of more primitive ones, like `And` and `Or`.

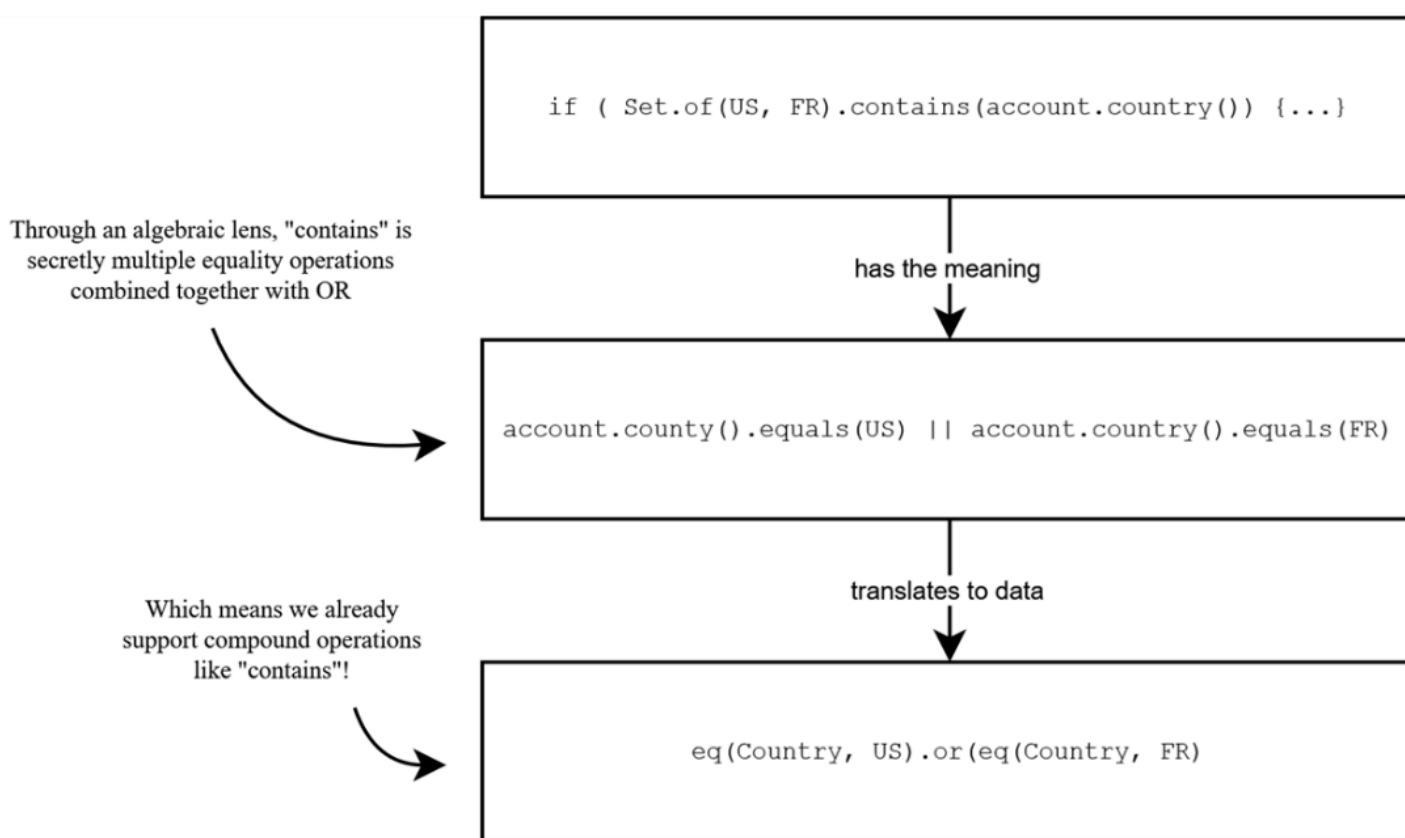


Figure 8.11 Viewing higher level operations like contains in terms of simpler primitives

And they're both binary operations, which means we can build new algebraic actions using the built-in stream tooling.

Listing 8.19 defining new ways of interacting with the algebra

```
static Rule any(Rule a, Rule... rest) {
    return Arrays.stream(rest).reduce(a, Or::new);    #A
}
static Rule all(Rule a, Rule... rest) {
    return Arrays.stream(rest).reduce(a, And::new);   #A
}

any(eq(COUNTRY, "US"), eq(COUNTRY, "FR"), eq(COUNTRY, "BE")); #B
eq(COUNTRY, "NA").or(eq(COUNTRY, "FR")).or(eq(COUNTRY, "BE")); #B

all(eq(COUNTRY, "US"), eq(SEGMENT, "Enterprise")); #C
eq(COUNTRY, "US").and(eq(SEGMENT, "Enterprise")); #C
```

#A We can create new rules by combining existing ones

#B Despite the different APIs, these are equivalent to each other

#C And these, too!

But we can do even better than this! Defining `eq(Country, ...)` over and over again is tedious. There's nothing that says we have to always work directly with `Rules`. We can interact with our Algebra however we want! Let's give ourselves a more convenient interface for building composite rules.

Listing 8.20 Adding more convenient ways of interacting with the algebra

```
static Rule contains(Attribute field, String opt1, String... rest) { #A
    return Arrays.stream(rest)                      #B
        .map(value -> eq(field, value))           #B
        .reduce(eq(field, opt1), Rule::or);       #B
}
```

#A We can tune the interfaces into our algebra however we want

#B Inside, we transform the input into data our algebra understands

Now we can write rules that are super compact and self-descriptive.

Listing 8.21 Increasing the clarity of our API

```
eq(COUNTRY, "US").or(eq(COUNTRY, "BE")).or(eq(COUNTRY, "FR"))
contains(COUNTRY, "US", "BE", "FR") #A
```

#A Sophisticated ideas without the boilerplate

What's really neat is that these new features automatically work with everything else in our algebra. This level of reuse usually takes herculean levels of planning and careful design, but it happens for free when building around a simple algebra. You end up getting more out of the system than you put into it.

We can freely mix and match these operations to build powerful rules.

Listing 8.22 Combinators combine with all other operations in our algebra

```
contains(COUNTRY, "US", "FR", "BE")          #A
    .and(eq(SEGMENT, "Public")).or(not(eq(REGION, "LATAM"))) #A
```

#A Our tiny algebra is suddenly able to express complex conditions very tersely

Now we have an algebra with enough power to cover all of our requirements. Here's the whole thing

Listing 8.23 The code so far

```
sealed interface Rule {  
    record Equals(Attribute field, String value) implements Rule {}  
    record And(Rule a, Rule b) implements Rule {}  
    record Or(Rule a, Rule b) implements Rule {}  
    record Not(Rule rule) implements Rule {}  
  
    static Rule eq(Attribute field, String value) {  
        return new Equals(field, value);}  
    static Rule not(Rule rule) {  
        return new Not(rule);  
    }  
    default Rule and(Rule other) {  
        return new And(this, other);  
    }  
    default Rule or(Rule other) {  
        return new Or(this, other);  
    }  
}
```

We've spent about 13 lines of code, and that bought us a tiny language inside of Java that allows us to tersely define business rules without all the boilerplate that comes with normal code.

But that's not actually the cool part yet. We're just getting started.

8.5 Variation I: Deciding what data means

We turned code into data, now we have to turn it back into code.

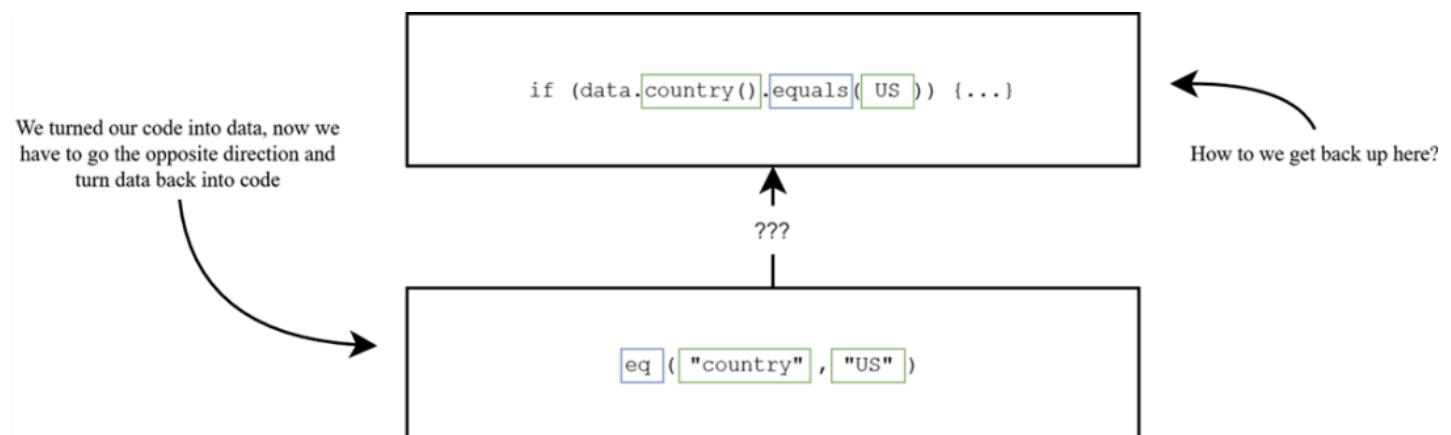


Figure 8.12 How do we get from data back to code?

It's up to us to decide what our data means.

Listing 8.24 a Interpreting the data

```
static ??? interpret(Rule rule, Account account) { #A
    return switch (rule) {
        case Eq(Attribute field, String value) -> ???; #B
        case Not(Rule r) -> ???; #B
        case And(Rule a, Rule b) -> ???; #B
        case Or(Rule a, Rule b) -> ???; #B
    };
}
```

#A We get to decide what the evaluation of our data produces

#B As well as what each of its data types mean

This means that we ask our usual “what does it denote?” question, but in reverse.

To start simple, we’ll map each data type in our algebra back to the same Boolean operations from which we originally extracted it.

We'll map our data back to the same string equality and boolean domain from which it was originally abstracted

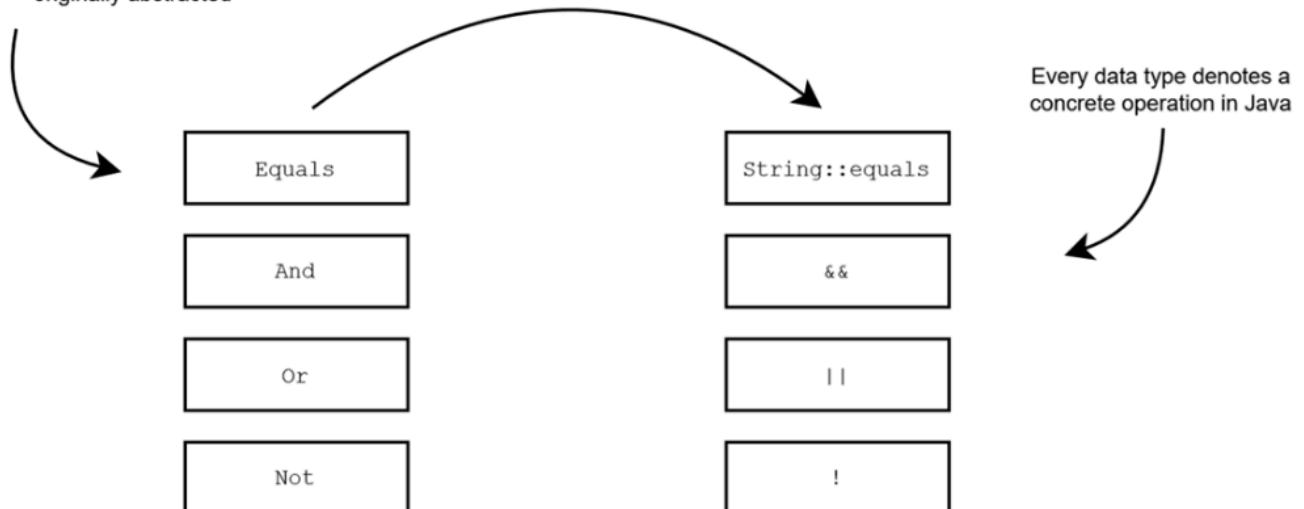


Figure 8.13 Translating data back into code

And once we know what our data denotes, the rest of the code falls into place. In fact, because we’ve modeled the rules as a sealed interface, the code basically writes itself. We just pattern match.

Listing 8.25 a Fleshing out the interpreter

```
static Boolean interpret(Rule rule, Account account) { #A
    return switch (rule) {
        case Equals(Attribute field, String value) -> ??? .equals(value); #B
        case Not(Rule r) -> ???;
        case And(Rule a, Rule b) -> ???;
        case Or(Rule a, Rule b) -> ???;
    };
}
```

#A Setting the return value to Boolean.

#B We know our equality check will look something like this, but we still have to define how we get from an Attribute enum back to a real value on an Account

We can similarly pattern match on the Attribute enum in order to get back to a concrete value on the Account data type.

Listing 8.26 Translating attribute names back to lookups against a record

```
static String get(Account account, Attribute attr) {          #A
    return switch (attr) {                                     #B
        case Attribute.REGION -> account.region().name();      #C
        case Attribute.COUNTRY -> account.country().name();     #C
        case Attribute.SECTOR -> account.sector().value();       #C
        case Attribute.SEGMENT -> account.segment().name();      #C
        case Attribute.CHANNEL -> account.channel().name();     #D
    };
}
```

#A Translating from data to code is straight forward

#B And it's type safe! Java will tell us if we miss any cases

#C For this first variation, we're interpreting our data in the "domain" of plain strings, so enums are stringified via name()

#D And we reach inside of SalesChannel to grab its String value

From here, it's a pretty literal translation back into code. Equals, the data type, becomes equals, the operation.

Listing 8.27 a basic interpreter

```
static Boolean interpret(Rule rule, Account account) {
    return switch (rule) {
        case Equals(Attribute field, String value) ->
            get(account, field).equals(value);  #A
        case Not(Rule r) -> ????
        case And(Rule a, Rule b) -> ????
        case Or(Rule a, Rule b) -> ????
    };
}
```

#A We can now evaluate and equality statement expressed as data!

The rest is just filling in the blanks. And, the data type, maps logical *and* (`&&`), the operation. Same for `Or` and `Not`. We just follow the denotations.

Listing 8.28 Following the data types

```
static Boolean interpret(Rule rule, Account account) {
    return switch (rule) {
        case Equals(Attribute field, String value) ->
            get(account, field).equals(value);
        case Not(Rule r) ->
            !interpret(r, account);                #A
        case And(Rule a, Rule b) ->
            interpret(a, account) && interpret(b, account); #B
        case Or(Rule a, Rule b) ->
            interpret(a, account) || interpret(b, account); #C
    };
}
```

#A We just follow what the data types denote. "Not" becomes "!" in Java

#B "And" denotes logical &&

#C "Or" denotes logical ||

And... that's it. Believe it or not, we're done. That's all it takes to build an interpreter for our data. No matter how complex our rules become, they can all be evaluated with this same logic.

Let's refactor our original implementation to use our new tools.

Listing 8.29 Refactoring to use our new tools

```
/**  
 * Rule #1  
 * All accounts in EMEA excluding those in Belgium, Austria,  
 * and France belong to SalesOrg=111  
 */  
static Optional<SalesOrgId> ruleForOrg111(Account account) {  
    Rule rule = eq(REGION, "EMEA")                      #A  
        .and(not(contains(COUNTRY, "US", "BE", "FR"))); #A  
    return interpret(rule, account)  
        ? Optional.of(new SalesOrgId("111"))  
        : Optional.empty();  
}
```

#A Code that reads exactly like its requirements

We've solved the ergonomic issue. Complex rules can be expressed without burning a bunch of code. But the really neat part is that this is still just ordinary data! And data is inherently portable. We can serialize it, save it, load it, pass it between systems – whatever we want!

Listing 8.30 Serializing to something like JSON is trivial because it's just data!

```
contains(Country, US, CA).and(Not(eq(Region, EMEA)))  
// out:  
{  
    "type": "AND",  
    "a": {  
        "type": "OR",  
        "a": {  
            "type": "EQ",  
            "field": "country",  
            "value": "US"  
        },  
        "b": {  
            "type": "EQ",  
            "field": "country",  
            "value": "CA"  
        }  
    },  
    "b": {  
        "type": "NOT",  
        "expr": {  
            "type": "EQ",  
            "field": "Region",  
            "value": "EMEA"  
        }  
    }  
}
```

That means our long-term goal of not writing these rules by hand is suddenly a lot more tractable. We could build a frontend that lets users construct rules in our

grammar. All powered by the same data.

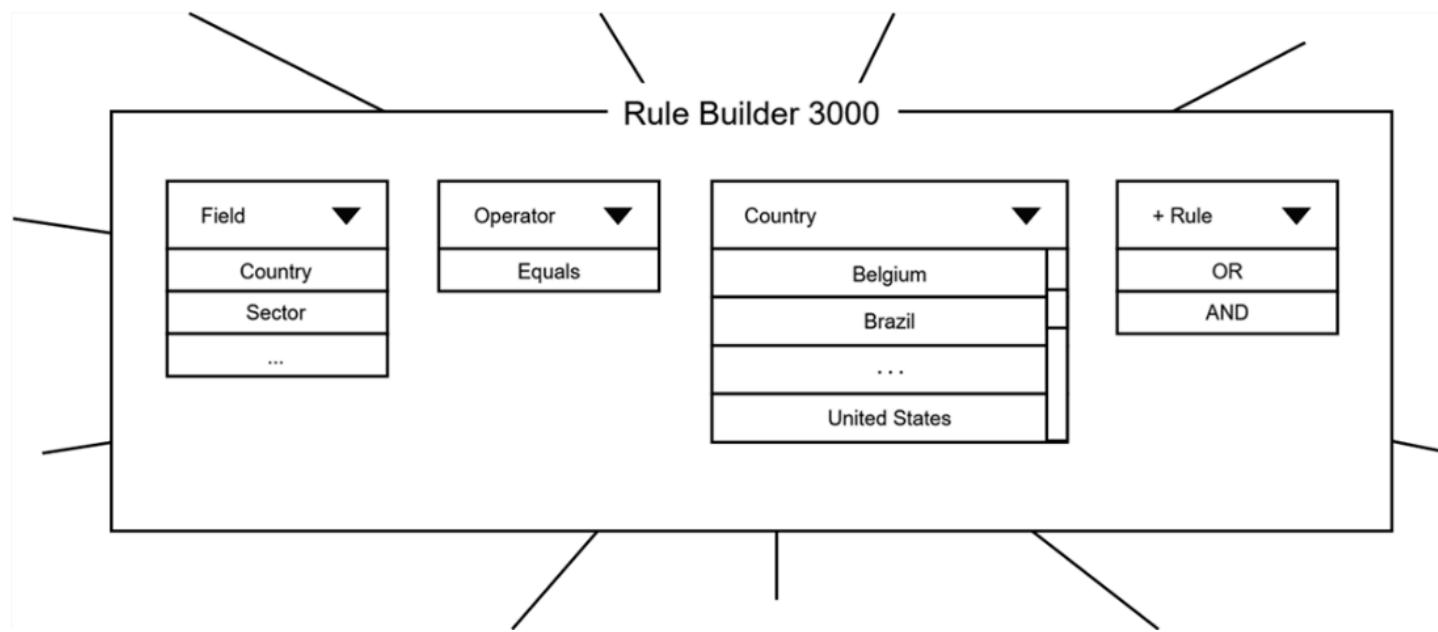


Figure 8.14 This same simple data model could power a fancy UI in the future

The only thing this doesn't solve is the original operational and debuggability woes. We still have no idea *why* a rule passes or fails.

To fix that, all we have to do is shift how we interpret what our data *means*.

8.6 Variation II: A more powerful interpreter for the same data

The trouble with Booleans is that they don't carry information about *why* they ended up in their current true or false state. So, let's fix that. We'll keep our data 100% the same, but simply *decide* that it denotes something new. We'll have our data map to a more sophisticated result type that can track why it's making decisions.

Like this:

Listing 8.31 A Boolean that tracks why it's in a certain state

```
record Result(  
    boolean matched, #A  
    String expected, #B  
    String found     #B  
){}  
#A Our core data type is still a boolean underneath the covers  
#B But we upgrade it to carry around why it's in its current state
```

Now we can map the meaning of each data type in our algebra to an operation against this new Result type.

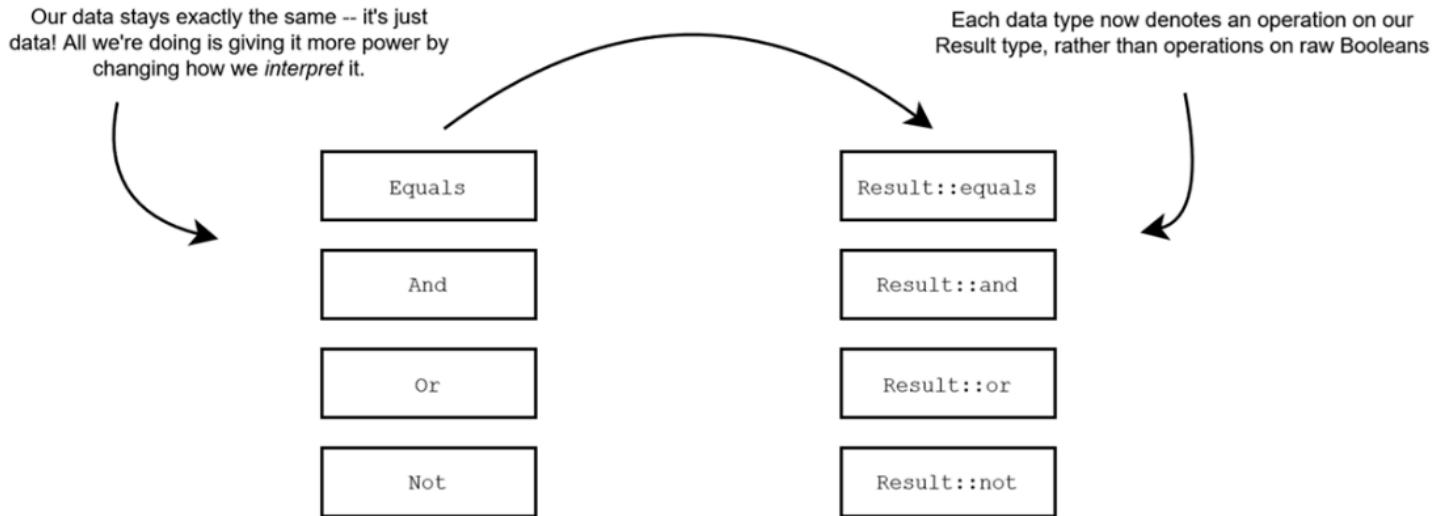


Figure 8.15 Changing what our data denotes

Now that we know what our data denotes, we can build a new interpreter to follow it.

Listing 8.32 Migrating our interpreter to the new Result type

```
static Result interpret(Rule rule, Account account) { #A
    return switch (rule) {
        case Equals(Attribute field, String value) -> {
            String found = get(account, field); #B
            boolean result = found.equals(value); #B
            yield new Result(
                result,
                format("%s=%s", field, value), #C
                format("%s=%s", field, found)); #C
        }
        case Not(Rule r) -> ???
        case And(Rule a, Rule b) -> ???
        case Or(Rule a, Rule b) -> ???
    };
}
```

#A Our new interpretation of the data returns a Result type, rather than a plain Boolean

#B The main logic is still exactly the same. We compare two strings.

#C But now we keep track of what those comparisons were and whether they matched what we expected

This small change yields something awesome. We get a nice little report of why we ended up with the answer we did.

Listing 8.33 JUnit style reporting on why a rule matched (or didn't!)

```
Rule rule = eq(Country, US);                      #A
interpret(rule, new Account(..., CountryCode.BE, ...)); #B

// output:
Result[                                #C
  matched=false,                      #C
  expected=(country=US),            #C
  found=(country=BE)]                #C
```

#A Nothing has changed in how we declare the rules. It's still just plain data.

#B It's our evaluation that has been enriched. We've imbued it with the ability to track the provenance of why it's making decisions.

#C We get a nice JUnit style report of what we expected versus what we found.

Best of all, *the complexity is hidden by the interpreter*. The data is exactly the same! All we do is say the rules we want applied. The code decides what to do with those rules.

The rest of the interpreter is more of the same. We follow the denotations. The only change from the previous interpreter is that we format provenance information as we evaluate.

Listing 8.34 finishing up the interpreter with AND, OR, and NOT

```
static Result interpret(Rule rule, Account account) { #A
    return switch (rule) {
        case Eq(Attribute field, String value) -> {...}
        case Not(Rule r) -> {                                #A
            Result res = interpret(r, account);           #B
            yield new Result(
                !res.matched(),                         #C
                format("not(%s)", res.expected),       #D
                res.found());                        #D
        }
        case And(Rule rule1, Rule rule2) -> {                  #E
            Result a = interpret(rule1, account);
            Result b = interpret(rule2, account);
            yield new Result(a.matched() && b.matched(),
                format("(%s AND %s)", a.expected, b.expected),
                format("(%s AND %s)", a.found, b.found));
        }
        case Or(Rule rule1, Rule rule2) -> {                  #E
            Result a = interpret(rule1, account);
            Result b = interpret(rule2, account);
            yield new Result(a.matched() || b.matched(),
                format("(%s OR %s)", a.expected, b.expected),
                format("(%s OR %s)", a.found, b.found));
        }
    };
}
```

#A Not just needs to invert whatever it finds

#B So we evaluate the rule

#C Then store the inverse on the result (note the ! on the boolean)

#D And lastly format our provenance information

#E AND and OR work exactly the same. Evaluate each expression and record the findings.

And... once again, that's it. We're done. That's all it takes to create a powerful interpreter that can evaluate rules while giving us back a report telling us what happened.

Listing 8.35 Full information on why we matched or didn't

```
Account account = new Account(  
    new AccountId("1234"),  
    Region.EMEA,  
    CountryCode.BE,  
    new Sector("Retail"),  
    Segment.Strategic,  
    SalesChannel.Reseller  
);  
  
Rule rule = (not(eq(Country, "US")).and(eq(Region, "AMER")))  
            .or(contains(Segment, "Enterprise", "Strategic"));  
  
interpret(rule, account);  
// output  
Result[  
  matched=false,  
  expected=((not(country=US) AND Region=AMER)  
            OR (Segment=Enterprise OR Segment=Strategic)),  
  found=((country=BE AND Region=EMEA)  
         OR (Segment=Enterprise OR Segment=Strategic))]
```

Now we can answer questions like “why did(n’t) you send Account X to Sales Org Y?” with ease. And best of all, we didn’t have to change our data! This is the power of decoupling. We’re free to continuously re-interpret the same data with increasingly powerful denotations.

But there are still some sharp edges in our algebra. Because we’re doing everything with raw strings, we’ve lost a lot of the compile time safety that data-orientation gives. It’s currently easy to create a non-matching rule by misremembering which values belong to which category, or mistyping a value. None of these will be caught until runtime.

8.7 Variation III: Adding Type Safety

String based APIs allow us to do nonsensical things like this:

Listing 8.36 The incorrect states our data currently allows

```
eq(COUNTRY, "Nacho Country"); #A  
eq(REGION, "US"); #B
```

#A until I'm elected and make some sweeping changes, this is not a valid country
#B This is surely to be a frustrating error. "US" sure sounds like a region, but it's completely invalid for the underlying enum.

That's annoying. Let's refactor so that the type system makes all of these errors impossible.

The first thing we have to fix is how we model the Equals data type. Right now, it maps *any* Attribute value to *any* String.

Listing 8.37 The current modeling of the Equals data type

```
record Equals(Attribute attribute, String value){} #A
```

#A Any attribute can be paired with any string

What we want is for `value` to be one of the actual data types on `Account`. If we're writing rules against a particular Segment, we should be able to use the `Segment` enum!

So, the first thing we can do is parameterize `Equals`. That allows us to say that `value` can be *any* type the user wants to supply.

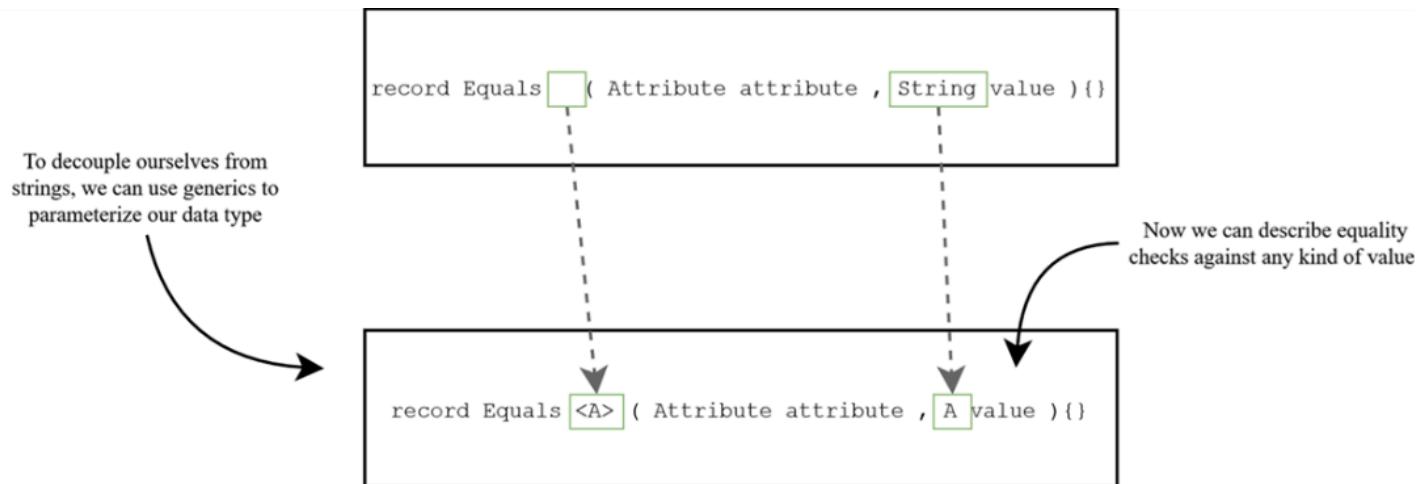


Figure 8.16 Parameterizing Equality with a generic type variable

This frees us from just passing around Strings. Now we can use the well modeled data types in our domain.

Listing 8.38 Using real types!

```
new Equals<>(COUNTRY, CountryCode.US);
new Equals<>(REGION, Region.EMEA);
```

However, it's not enough yet. We can still supply the *wrong* type.

Listing 8.39 Whoops!

```
new Equals<>(COUNTRY, Region.EMEA);    #A
new Equals<>(REGION, CountryCode.US); #A
```

#A Both totally invalid

What we need to do next is come up with a way to make these two input types *depend* on each other. If we're writing a rule against the `Segment` attribute, we should only be allowed to supply values from the `Segment` enum.

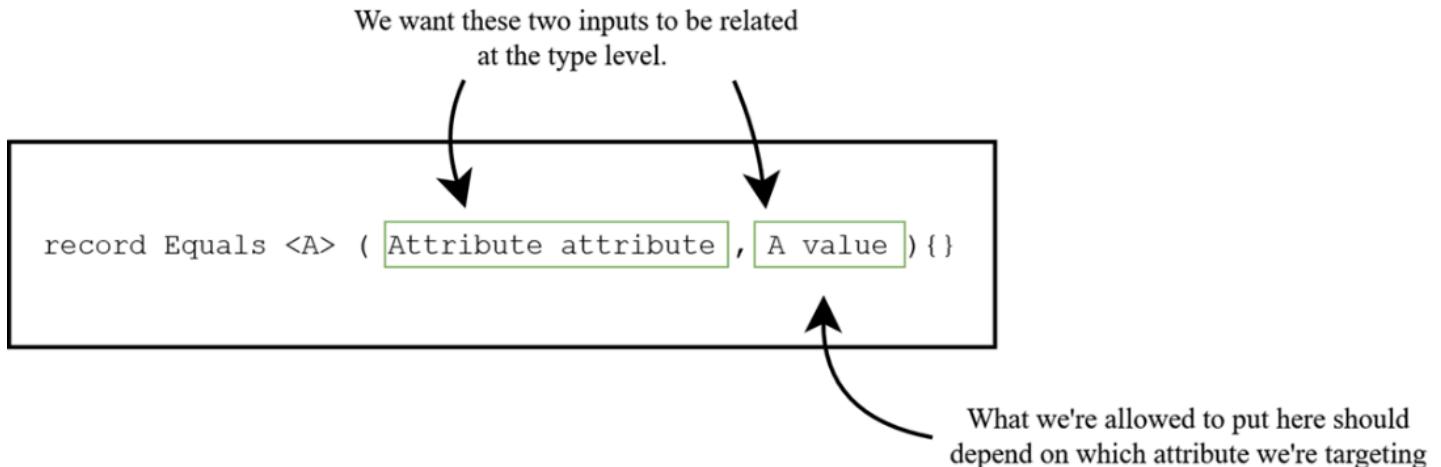


Figure 8.17 Somehow, we have to relate these two input values at the type level

There's no way to do that with our current setup. Enums cannot be made generic.

Listing 8.40 This won't work

```
enum Attribute<A> { #A
    ...
}
```

#A Not allowed in Java

What we can do to get around this problem is *wrap* that enum in something that *can* be generic. We can create a record that parameterizes Attribute with extra type information.

Listing 8.41 Creating a wrapper type with extra type information

```
record Attr<A>(
    Attribute value
){}
```

Now we can use this new parameterized record on our Equals model to force that all the types line up when we're creating rules.

Now Java can enforce that all of these
be the same type!

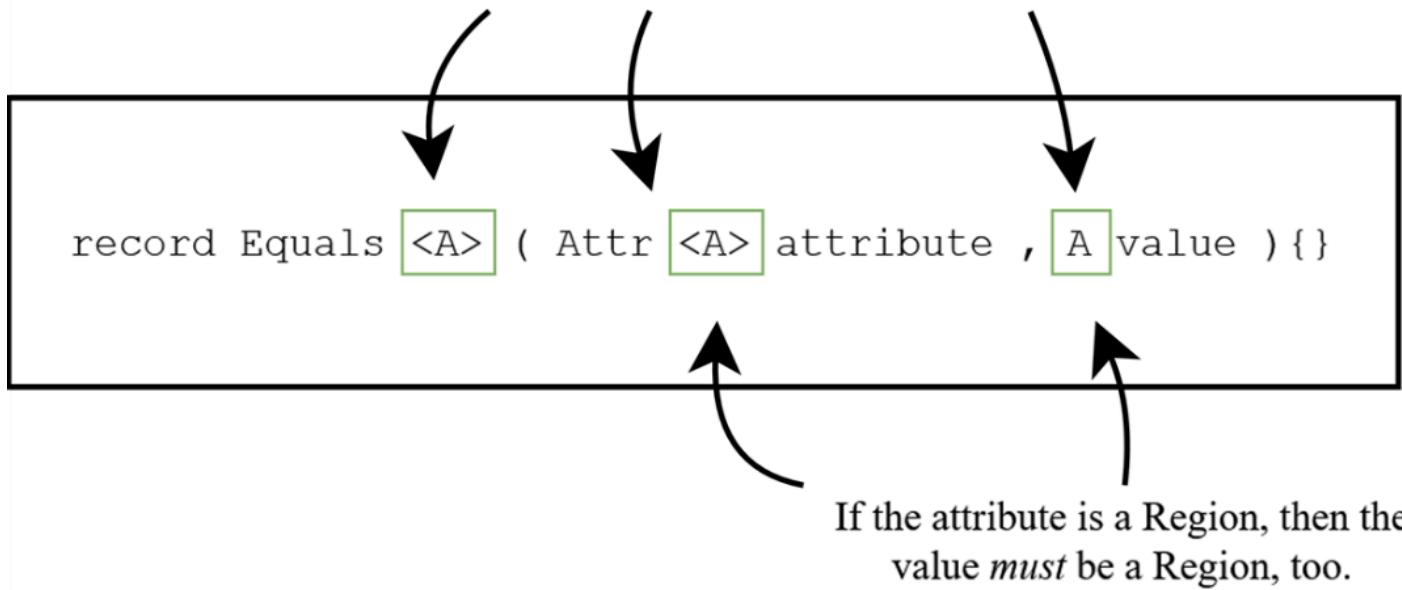


Figure 8.18 Forcing that types all be the same

By passing `Attr<A>` rather than the vanilla `Attribute` enum, we can make sure that the region attribute receives values from the `Region` enum, and segment attributes get valid `Segment` values, and so on for all the other types.

Listing 8.42 Type safety! Hooray!

```
static Attr<SalesChannel> channel = new Attr<>(Attribute.CHANNEL); #A
static Attr<Region> region = new Attr<>(Attribute.REGION);           #A
// etc...

new Equals<>(channel, SalesChannel.Reseller);    #B
new Equals<> (region, Region.EMEA);               #B

new Equals<>(channel, Region.EMEA) // Nope!      #C
```

#A Statically defining instances for each Attribute with the extra type info we need

#B Now only valid values can be supplied to valid attributes

#C If you try to mix invalid types, Java won't even compile. Pretty neat!

We're almost to the finish line. There's one last thing we have to do to make this work. Despite the fact that it looks like it should work, we don't have enough type information to convince Java that an `Attr<Region>` can be safely turned into an actual instance of `Region`.

Listing 8.43 This doesn't work

```
static <A> A get(Account account, Attr<A> attr) {  
    return switch (attr.attribute()) {  
        case Region -> account.region(); #A  
        case Country -> account.country(); #A  
        case Sector -> account.sector(); #A  
        case Segment -> account.segment(); #A  
        case Channel -> account.channel(); #A  
    };  
}
```

#A All of these produce type errors.

The problem is that our generic type variable `A` can be anything, but we're trying to compress it back down to one of the few types that exist on an `Account`. Java is correctly scolding us for doing something that doesn't make sense.

Here's the final bit of type system sleight of hand that makes this all work: we can make the `Attr<A>` type carry evidence that we have *some* way of going from an `Account` to the target type we're creating. Thus, we prove to the type system that our mapping is sound.

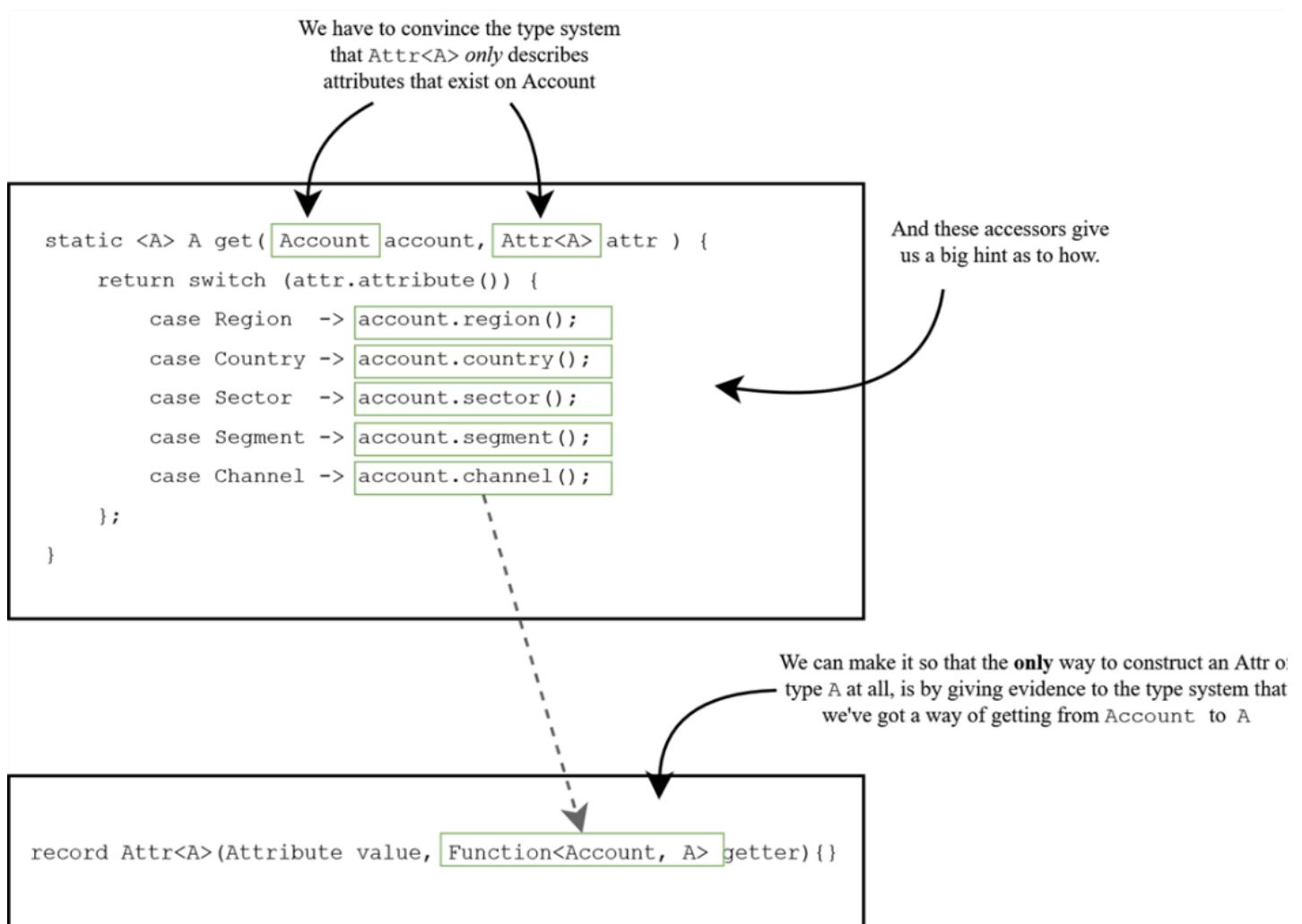


Figure 8.19 Giving concrete evidence to the type system that we know what we're doing

That lets us update our type safe attributes with evidence showing Java we know how to access them.

Listing 8.44 Adding evidence that we know how to access our data

```
record Attr<A>(Attribute attribute, Function<Account, A> getter){} #A

Attr<SalesChannel> channel
    = new Attr<>(Attribute.Channel, Account::channel); #B
Attr<Region> region
    = new Attr<>(Attribute.Region, Account::region); #B
```

#A The only way to create an attribute now is to supply evidence that we know how to get from an Account to the type we're defining

#B Method reference makes it easy to supply the evidence functions

Now that we carry around the knowledge of how get things out of an Account, we don't even need a special function to switch on the `Attribute` enum anymore. We can do the conversion directly in our interpreter

Listing 8.45 Refactoring to simplify how we lookup values

```
attr.getter().apply(account) #A
```

#A this is now all we need to convert from data to concrete code

Now that everything is generic, we can no longer specify concrete types in our interpreter. We could technically be processing *any* type. So, our pattern match becomes a wildcard.

Listing 8.46 Comparing the change from hard coded to generic

```
case Equals(Attribute attr, String value) -> ... #A

case Equals<?> eq -> ... #B
```

#A The old hard coded version that only worked with strings

#B The new generic version that works with any type. The ? is a wildcard standing for "any type."

But there's another way we could do this, too. One of the really cool features of Java 21's record patterns is that we can pattern match even when we have no idea what the types of the arguments are! We can still assign them to variables using Java's `var` keyword.

Listing 8.47 Pattern matching without specifying concrete types

```
case Eq(var attr, var value) -> ... #A
```

#A We can use record patterns even when we don't know the types!

It's a small improvement, but a really nice one.

Now that we know how to pattern match wildcarded data, let's finish up the rest of the changes to the equality check logic.

Listing 8.48 Finish up the equality implementation

```
static Result interpret(Account account, Rule rule) {
    return switch (rule) {
        case Eq(var attr, var value) -> {
            var found = attr.getter().apply(account); #A
            boolean result = found.equals(value);      #B
            yield new Result(result,
                format("%s=%s", attr.attribute(), value), #C
                format("%s=%s", attr.attribute(), found)); #C
        }
    }
}
```

#A First we lookup the value inside of the account. Note that we use var again as a placeholder since we don't know the type

#B Then the equality check. This is guaranteed to be safe thanks to all the information we put into the type system

#C And then just attaching the provenance info as usual.

That's it! The other branches are all unchanged.

Here's the best part: this is still just data! All the extra type information is just for our convenience as programmers. It's equally easy to serialize with a few annotations or a couple of helper functions. We can drop the type information on the way out the door and add it back on the way in.

8.8 What did this cost us?

We built a whole interpreter by hand rather than installing a dependency. How much code did this take? Here's the whole thing in its type safe glory:

Listing 8.49 The full type-safe data model and interpreter

```
enum Attribute {REGION, COUNTRY, SECTOR, SEGMENT, CHANNEL}
record Attr<A>(Attribute attribute, Function<Account, A> getter){}
record Result(Boolean matched, String expected, String found){}

sealed interface Rule {
    record Equals<A>(Attr<A> field, A value) implements Rule {}
    record Or(Rule a, Rule b) implements Rule {}
    record And(Rule a, Rule b) implements Rule {}
    record Not(Rule rule) implements Rule {}
    default Rule or(Rule b) {
        return new Or(this, b);
    }
    default Rule and(Rule b) {
        return new And(this, b);
    }
}

static Attr<SalesChannel> channel =
    new Attr<>(Attribute.CHANNEL, Account::channel);
static Attr<Sector> sector =
    new Attr<>(Attribute.SECTOR, Account::sector);
static Attr<CountryCode> country =
    new Attr<>(Attribute.COUNTRY, Account::country);
static Attr<Region> region =
    new Attr<>(Attribute.REGION, Account::region);
static Attr<Segment> segment =
    new Attr<>(Attribute.SEGMENT, Account::segment);

static Result interpret(Account account, Rule rule) {
    return switch (rule) {
        case Equals(var attr, var value) -> {
            var found = attr.getter().apply(account);
            boolean result = found.equals(value);
            yield new Result(result,
                format("%s=%s", attr.attribute(), value),
                format("%s=%s", attr.attribute(), found));
        }
        case Not(Rule r) -> {
            Result res = interpret(account, r);
            yield new Result(!res.matched(),
                format("not(%s)", res.expected),
                res.found());
        }
        case Or(Rule rule1, Rule rule2) -> {
            Result a = interpret(account, rule1);
            Result b = interpret(account, rule2);
            yield new Result(a.matched() || b.matched(),
                format("(%s OR %s)", a.expected, b.expected),
                format("(%s OR %s)", a.found, b.found));
        }
        case And(Rule rule1, Rule rule2) -> {
            Result a = interpret(account, rule1);
            Result b = interpret(account, rule2);
            yield new Result(a.matched() && b.matched(),
                format("(%s AND %s)", a.expected, b.expected),
                format("(%s AND %s)", a.found, b.found));
        }
    }
}
```

```
};  
};
```

About 60 lines of code. No dependencies. Just pure Java.

8.9 B—but! It doesn't do everything!

That's OK! Too often, we software people get wrapped up in "more." More power. More features. We pull in unnecessary tools just in case we might need them one day.

I hate to even call what we did here an "interpreter," because it anchors it on being something more important than it is. I feel similarly pigeonholed by calling it a DSL. The problem with both of those descriptions is that they kick off all kinds of dangerous ideas in the mind of the developer. Once you're "building an interpreter," it's suddenly something Big and Important. You *have* to add more features, because the list of things it doesn't do far exceeds the list of things it does. And if you're measuring it in terms of raw capabilities, nothing we build by hand will ever be "complete." There will always be use cases it doesn't handle, and thus you "have to" pull in that library, or framework, or whatever.

Instead, you should think of this as just normal, everyday business code. It doesn't do anything other than meet its requirements, and that's OK. We're not building a framework or the next Big Super Important Enterprise Rules Engine, we're just writing some code.

This distinction is important, because the lines between "interpreter," "DSL," and "just writing some code" all start to blur when you focus on representing the core ideas in your domain as data. Quite a few problems fall into this "shape." And it's OK to use this shape to solve *just* those problems, rather than try to solve every problem that might ever exist.

In fact, we've been secretly building these little mini-interpreters since the very first chapter! Remember this?

Listing 8.50 Plain data that we "interpreted" later on

```
sealed interface RetryDecision {  
    record RetryImmediately(...) implements RetryDecision {}  
    record ReattemptLater(...) implements RetryDecision {}  
    record Abandon(...) implements RetryDecision {}  
}
```

The more you model with data, the more you stumble into this pattern where you describe what you want to do in one place and have code that handles it elsewhere.

8.10 Data is flexible in a way that code isn't

Time has passed in our imaginary system. The usual chaos took place. Some rules were added through the "official" process, others via ad-hoc requests and conversations. Some, well, we're not really sure why they're there. We should probably write better commit messages. To "increase trust," management decides that we will publicly document every rule in our system so that stake-holders can review them.

How hard this is depends on our earlier design decisions.

Image for a second that we stuck with the original approach of writing our rules as code.

Listing 8.51 Rules as code

```
Set<CountryCode> included = Set.of(BE, AU, FR);      #A
included.contains(account.country())                 #A
  && (account.segment().equals(Public))             #A
    || !account.region().equals(LATAM))
...
...
```

#A How would we extract this and turn it into documentation?

What the heck would we do? The information is “stuck” in the code. We can see the rules with our eyeballs, but we have no programmatic mechanism for accessing it (beyond, say, parsing and crawling the raw AST). It is very hard to write a program that extracts semantic information from the source code of another program.

That means we’d surely be stuck writing the documentation by hand. And that means our documentation will be wrong. Maybe not initially, but over time the two will drift apart.

Now let’s flip back to the current world where we’re our rules are expressed as data.

Listing 8.52 Rules as data

```
contains(country, US, BE, FR)
  .and(eq(segment, Public).or(not(eq(region, LATAM))));
```

How will we generate documentation from this? Any way we want! This new requirement is a total non-problem. Data is scrutable in a way that code isn’t. Data can be interrogated and studied and processed and transformed – because it’s just data! It is trivially easy to derive new information from it because it’s not locked away in code.

So, let’s use these data powers and solve the requirement. A first stab might look like a simplified version of the interpreters we’ve already built:

Listing 8.53 Generating documentation automatically

```
static String document(Rule rule) {
  return switch (rule) {
    case Equals(var field, var value)
      -> format("%s = %s", field, value);
    case Not(Rule r)
      -> format("not (%s)", document(r));
    case Or(Rule a, Rule b)
      -> format("(%s or %s)", document(a), document(b));
    case And(Rule a, Rule b)
      -> format("(%s and %s)", document(a), document(b));
  };
}
```

A few lines of code and we’ve got a documentation pipeline that’s guaranteed to be in sync with the rules, because it’s derived directly *from* the rules. Not bad – although its output might be too spartan for our target audience.

Listing 8.54 Easy to generate, but not exactly easy to read

```
((COUNTRY = US or COUNTRY = BE) or COUNTRY = FR) #A
and (SEGMENT = public or not (REGION = LATAM))) #A
```

#A So many parentheses! What is this -- Lisp!?

Can we do better? Of course! We can put data in any shape we want. Hidden inside of that soup of parentheses is a Tree that we can visualize for our users. You can see this tree if you carefully line up the `toString()` output of our Record. Let's set it free and add some pizzazz to our documentation.

Listing 8.55 A tree laying on its side

```
And[
  a=Or[
    a=Or[a=Equals[field=COUNTRY, value=US],      #B
          b=Equals[field=COUNTRY, value=BE]],       #B
    b=Equals[field=COUNTRY, value=FR]],           #B
  b=Or[
    a=Equals[field=SEGMENT, value=public],        #B
    b=Not[a=Equals[field=REGION, value=LATAM]]]   #B
]
]
```

#A The root of the tree

#B Leaf nodes

There is no shortage of libraries for visualizing hierarchical data. Mermaid and GraphVis are popular ones. Each has its own specifics, but they generally work the same: we specify the nodes and edges, it renders a pretty graph.

Listing 8.56 A quick overview of GraphVis' adjacency list as text

```
// Nodes
node_123 [label="Hello"] #A
node_456 [label="World"] #B

// Edges
node_123 -> node_456 #C
```

#A Nodes are identified by an ID

#B The label attribute controls what's displayed when rendered

#C Edges are described as an adjacency list. The arrow (->) denotes that Node_123 is connected to node_456

So, we need a way of transforming our Rules (data) into the GraphVis language (also data). We'll start with a few helpers for generating the IDs, nodes, and edges.

Listing 8.57 A few helper functions to transform Rules into GraphVis lingo

```
static String id(Rule rule) {                                     #A
    return "node_" + Integer.toHexString(rule.toString().hashCode()); #A
}

static String edge(Rule from, Rule to) {                      #B
    return format("%s -> %s", id(from), id(to));          #B
}

static String node(Rule rule) {
    return switch (rule) {
        case Equals(var field, var value)                   #C
            -> format("%s [label=\"%s=%s\"]", id(rule), field, value); #C
        default
            -> format("%s [label=\"%s\"]",                  #C
                        id(rule),                         #C
                        rule.getClass().getSimpleName()); #C
    };
}
```

#A The generates a unique(ish) ID. A proper SHA would avoid collisions, but this is good enough for our purposes.

#B Edges are simple. An ascii arrow (->) between the two ids

#C To make the visualization pretty, we give the nodes different styling rules depending on if they're a leaf node or a branch.

We could tackle converting Rules into GraphVis in any number of ways, but for compactness, we'll do it all in one go and generate the nodes and edges while we traverse.

Listing 8.58 Traversing to generating nodes and edges

```
@SafeVarargs
static <A> Stream<A> concat(Stream<A>...ys) { #A
    return Arrays.stream(ys).reduce(Stream.of(), Stream::concat);
}
static Stream<String> nodesAndEdges(Rule rule) { #B
    return switch (rule) {
        case Equals e -> Stream.of(node(e));
        case Not not -> concat(
            Stream.of(node(not)),
            Stream.of(edge(not, not.a())),
            nodesAndEdges(not.a())));
        case Or or -> concat(
            Stream.of(node(or)),
            Stream.of(edge(or, or.a())),
            Stream.of(edge(or, or.b())),
            nodesAndEdges(or.a()),
            nodesAndEdges(or.b())));
        case And and -> concat(
            Stream.of(node(and)),
            Stream.of(edge(and, and.a())),
            Stream.of(edge(and, and.b())),
            nodesAndEdges(and.a()),
            nodesAndEdges(and.b())));
    };
}

static String toGraphVis(Rule rule) { #C
    return String.format("""
        digraph Rule {
            rankdir=TD;
            %s
        }"", String.join("\n\t", nodesAndEdges(rule).toList()));
}
```

#A Helper for combining an arbitrary number of streams

#B Traverses over the tree of Rules and generates a stream of Nodes and Edges.

#C Finally, we stitch it all together and generate the contents of the GraphVis file

Another handful of code and we've moved our data into a new world and imbued it with a new power.

Listing 8.59 Different views of the same data

```
// Data in code
contains(COUNTRY, US, BE, FR)
    .and(eq(SEGMENT, "public").or(not(eq(REGION, LATAM))))  
  
// Data in GraphVis
digraph Rule {
    rankdir=TD;
    node_9d54a989 [label="And"]
    node_9d54a989 -> node_cf7ac4e3
    node_9d54a989 -> node_141bf586
    node_cf7ac4e3 [label="Or"]
    node_cf7ac4e3 -> node_a4bf6d1d
    node_cf7ac4e3 -> node_85ef509a
    node_a4bf6d1d [label="Or"]
    node_a4bf6d1d -> node_85ef8908
    node_a4bf6d1d -> node_85ef4003
    node_85ef8908 [label="COUNTRY=US"]
    node_85ef4003 [label="COUNTRY=BE"]
    node_85ef509a [label="COUNTRY=FR"]
    node_141bf586 [label="Or"]
    node_141bf586 -> node_930ad8e0
    node_141bf586 -> node_a5e4027c
    node_930ad8e0 [label="SEGMENT=public"]
    node_a5e4027c [label="Not"]
    node_a5e4027c -> node_b83f3b85
    node_b83f3b85 [label="REGION=LATAM"]
}
```

A final quick call to the library and we get a beautiful visualization of our rules that we can embed into our documentation.

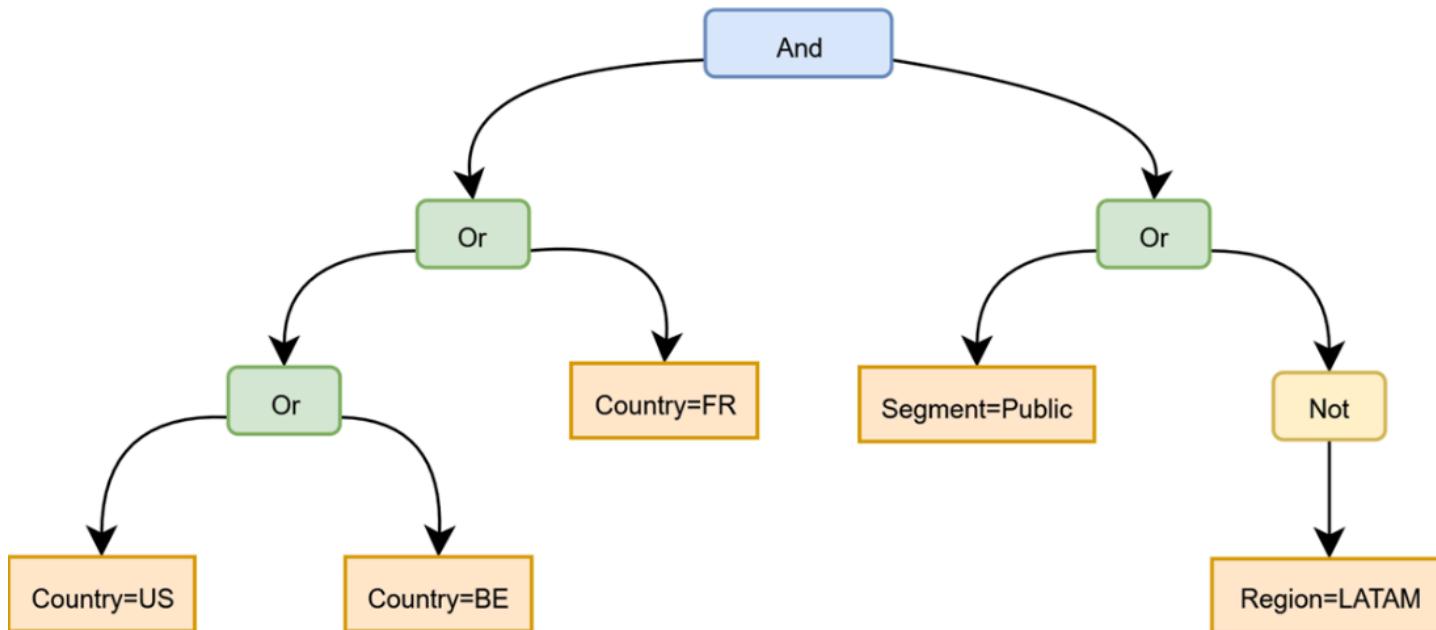


Figure 8.20 Visualizing rules

And we don't have to stop there. The ability to interrogate and interact with our data is what makes it so uniquely flexible. We can just keep adding new ways of interpreting it. Entire suites of tooling can be built around generating metadata about our data. With data, we're ready for anything the world can throw at us.

Speaking of which...

8.11 Extending the algebra:

Due to power grabs and moat building, the sales orgs have decided that territories should be split between owners based on the account's total spend. "Large" accounts should be handled by a different sales org than "small" accounts. They want to be able to express rules like "any account with a total spend larger than \$5 million belongs to Joe's org."

Listing 8.60 New information on the Account

```
record Account(
    AccountId accountId,
    Region region,
    CountryCode country,
    Sector sector,
    Segment segment,
    SalesChannel channel,
    USD totalSpend #A
){}
```

#A New!

We can extend our algebra to represent comparisons like GreaterThan and LessThan.

Listing 8.61 Adding a new option to our algebra

```
sealed interface Rule {
    record Equals(...) {
        record GreaterThan<A extends Comparable<A>>(Attr<A> attr, A value) #A
            implements Rule{}
        record LessThan<A extends Comparable<A>>(Attr<A> attr, A value)      #A
            implements Rule{}
    }
}
```

#A Adding the ability to describe comparisons between attributes. Note that we require A to be Comparable

However, to get this to work, we have to do some type-level shenanigans. The problem is related to the wildcards we talked about in section 8.7. Java doesn't "see" the underlying generic types while we're pattern matching in a switch expression, it only knows that they could be *any* type.

Listing 8.62 The problem: Java can't tell these are the same kind of Comparable

```

static Result eval(Rule rule, Account account) {
    return switch (rule) {
        case Eq(var field, var value) -> {...}
        ...
        case GreaterThan<?> gt -> {
            Comparable<?> found = gt.attr().getter().apply(thing); #A
            Comparable<?> expected = gt.value(); #A
            boolean result = found.compareTo(value) > 0; // ERROR #B
        }
    };
}

```

#A Unlike our equals implementation which could “see” an object, Java “sees” two Comparable instances of possibly any type

#B And it can’t tell that they’re the same, so we get a compile error if we try to compare them

So, we have to give it a nudge in the right direction. We can do that by performing the comparison operation in its own method with lots of type information.

Listing 8.63 Reminding Java that we’re all speaking the same language here

```

<A extends Comparable<A>> int compareTo(
    GreaterThan<A> gt, Account thing) { #A
    A found = gt.attr().getter().apply(thing);
    return found.compareTo(gt.value());
}

```

#A This gives java the needed type information to get us out of Comparator<?> and into Comparator<A>; where everything is the same

Once we have that, it’s business as usual. We just plug the types into the interpreter, and away we go.

Listing 8.64 Adding support for GreaterThan and LessThan

```

static Result eval(Rule rule, Account account) {
    return switch (rule) {
        case Eq(var field, var value) -> {...}
        ...
        case GreaterThan<?> gt -> {
            Comparable<?> found = get(account, gt.attr()); #A
            Comparable<?> expected = gt.value(); #A
            boolean result = compareTo(gt, account) > 0; #B
            yield new Result(result,
                format("%s>%s", gt.attr().attribute(), expected),
                format("%s=%s", gt.attr().attribute(), found));
        }
    };
}

```

#A Out here, we still see them as wildcards, which is OK, because we’ll only call toString on them
#B Inside of this function call is where they’ll be “seen” as types which are all on the same plane

And... that’s it! Now we can write rules that compare data, and get back a full report of why it did or didn’t match.

Listing 8.65 Writing rules that compare data

```
eq(Region, EMEA).and(gt(USD.of(10_000_000.0)))  
  
Result[matched=false,                                     #A  
      expected=(Region=EMEA AND Sales>=10,000,000.0), #A  
      found=( Region=EMEA AND Sales=500,000.00)]       #A
```

#A Pretty cool, right?

8.12 Wrapping up

Modern Java is lean. Modeling operations as data allows us to build powerful interpreters into the language itself. What used to require complex design patterns and layers of indirection in order to approximate are now available directly under our fingertips. Records and pattern matching have changed the cost of building in Java.

Next up, we're going to zoom out to the application level and look at how we can use all the tools we've learned to keep our programs safe against the outside world.

8.13 Summary

- Records and pattern matching have changed the cost of building custom DSLs in Java
- In addition to nouns, “things we know,” and decisions, data can also model *operations*
- Modeling operations as data decouples what we want to do from how we do it
- It also decouples *when* we do it. Data describes operations we’ll evaluate *later*
- Operations as data are just data! They can be stored, serialized, transformed, and transmitted like any other data
- Expressing operations as data types allows us to build algebras on top of that data
- Complex algebraic operations can be built from simpler primitives (like `And` and `Or`)
- Algebras aren’t “stuck” with one data type; we can build layers that transform more convenient data types into our algebra
- Designing around an algebra yields compositional APIs without any effort on our part
- Interpreters are just denotations in reverse. We decide what the data means
- Data can be interpreted however we want (and in more than one way!)
- The line between interpreter, DSL, and “just writing code” is blurry in the world of data-oriented programming.
- The more well modeled your data, the more it will fall into interpreter-like shapes
- It’s OK to build small “interpreters” that solve the problem at hand and nothing else!

9 Refactoring towards Data

This chapter covers

- Refactoring towards data orientation
- Drawing boundaries with data
- Writing good utility functions

Most codebases won't be written in a data-oriented fashion (at least not yet). They'll be the usual mix of object-oriented and imperative styles. Worse, no one will hand us "data-oriented requirements" or tell us to implement something in a "data-oriented way." It's up to us to tug codebases in the direction we want them to go. This means refactoring. We have to find the data hidden in everyday code and bring it to the surface.

This chapter is a guided tour of that refactoring process. It's the culmination of all the techniques we've explored throughout this book. We're going to take a messy object-oriented(ish) codebase and slowly refactor it towards a clean, data-oriented one. Trading objects for data makes these refactors small and incremental. We don't have to solve everything at once. The right data type makes all the difference.

Throughout this process, we'll flex the modeling and correctness skills we've learned, but we'll also explore new ways of viewing what our data modeling means. We're going to climb to the top of the ivory tower to explore something called the Curry-Howard correspondence. This vantagepoint will enable us to draw powerful boundaries that move even the messiest codebase towards one that's simple, descriptive, and correct.

There's a lot to explore, and it all begins with data.

Let's get started!

9.1 A messy starting point

You've inherited the company's legacy e-commerce service! It's standard stuff: Customers browse our catalog, add items to their cart, and checkout when they're ready to buy.

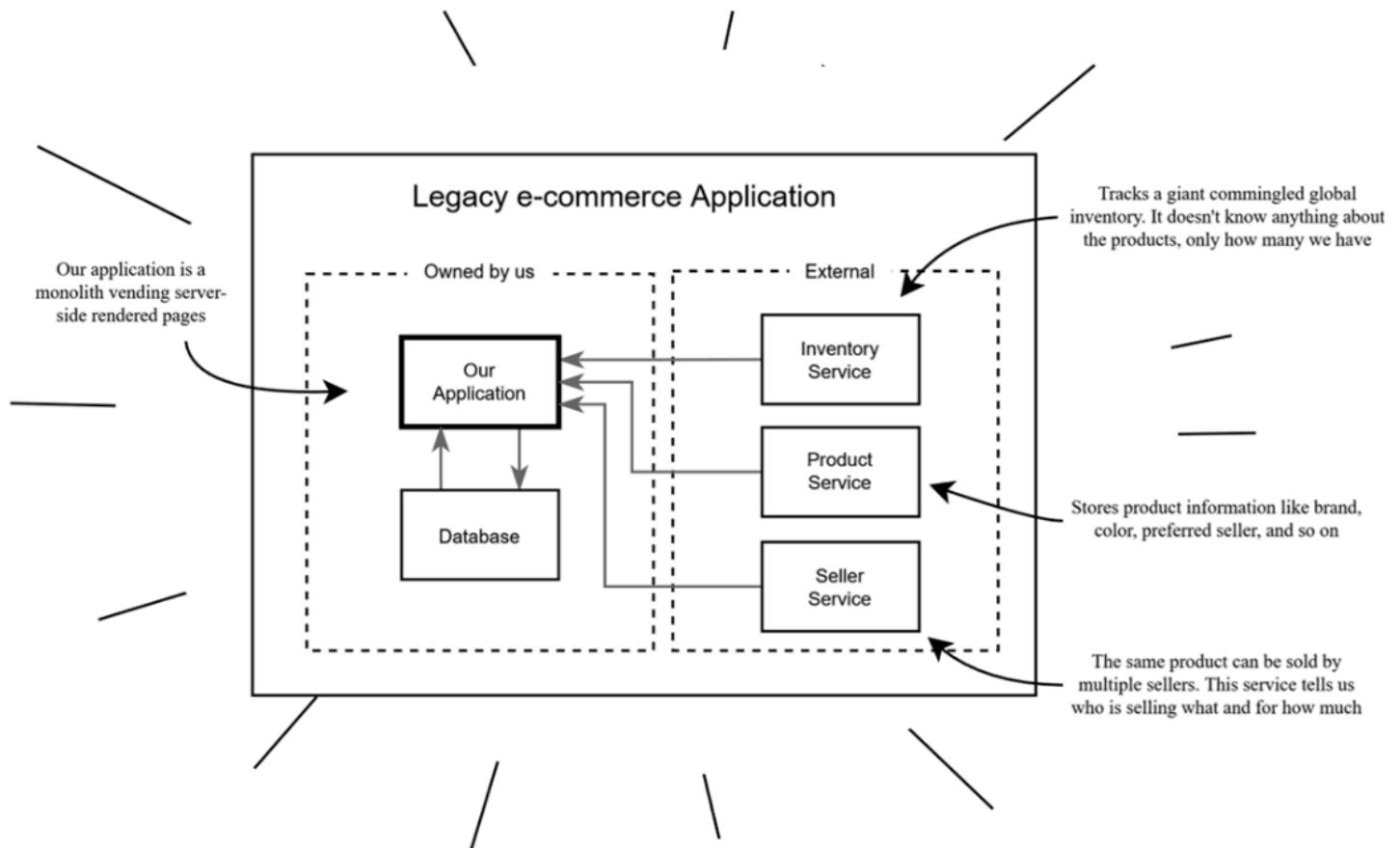


Figure 9.1 The application

Over the years, project managers have come up with all kinds of features to “improve customer experience.” One of the big ones was enhancing the cart with a “Saved for Later” section. This section lets us helpfully remind the customer to buy that thing they didn’t buy that one time. Spend your money! Consume! Buy it now!

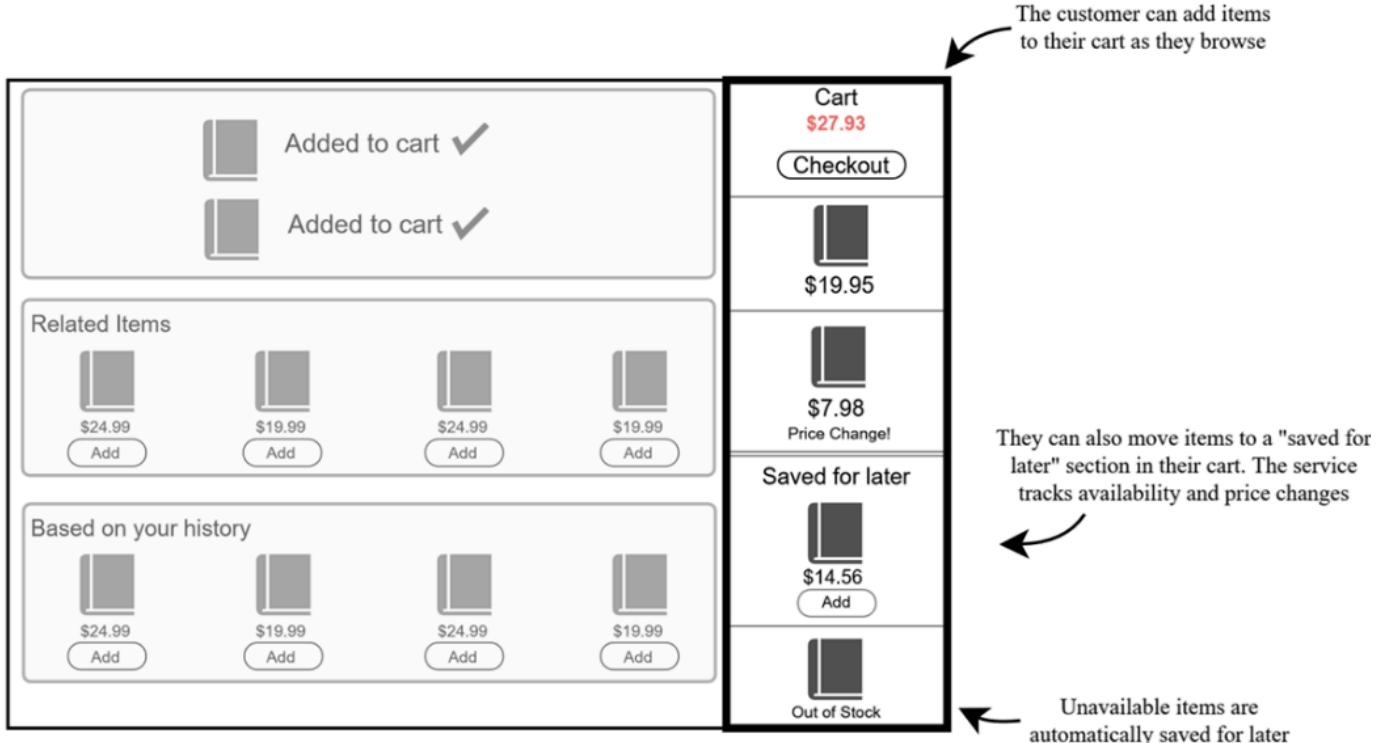


Figure 9.2 Customers (and our system) can move items from their “active” cart into “saved for later”

The steady march of features has turned the act of adding an item in a cart into a complicated dance of inventory checks, state management, upselling, and so on. With that complexity has come bugs and a big slowdown in velocity. It’s our job to refactor it into something better.

Here are the last known requirements.

Table 9.1 Requirements for adding items to a cart

ID	Requirement
1	Customers shall have an Active and Saved cart
2	Customers shall be able to add items to their Active cart
3	When a customer adds items to their cart
3.1	If the item is already in the Active cart, the service shall add their quantities
3.2	If the item is already in the Saved cart, the service shall add their quantities and move the item to Active
3.3	If the item is not in either cart, it shall be added to Active
4.0	The system shall notify customers when items in their Active or Saved carts change prices
5.0	The system shall automatically move unavailable Active items to Saved

This system follows the same style we used in Chapter 05: one heavily inspired by popular web frameworks. It’s object oriented in some places and imperative in others. It uses ORMs and web services. We hop into the story years into its development. The original team is long gone. The one after that, too.

The service is designed around two primary Entity types: `ShoppingCart` and `CartItem`. These classes do everything. They’re the database model, and the application model, and what’s used to render the front end.

In short, there are some boundary issues in the codebase.

Listing 9.1 The main entity classes

```
@Entity
public class Cart {          #A
    @Id
    String cartId;
    List<CartItem> active;  #B
    List<CartItem> saved;   #B
    BigDecimal subtotal;
    boolean hasPriceChanged;
    ...
    List<CartItem> recommended;
    boolean hasGiftOptions;
    // plus dozens more attributes
    // and numerous methods for modifying the cart
}

public class CartItem {
    @Id String id;           #C
    String productId;         #D
    String sellerId;          #D
    long desiredQty;          #E
    long availableQty;        #F
    BigDecimal latestPrice;   #G
    BigDecimal lastNotifiedPrice; #H
    Details productDetails;   #I
    ...
    // and another dozen or so here, too
    ...
}
```

#A The Cart is everything. The presentation, the business logic, the persistence. It's speckled with dozens of fields that get used, uh... somewhere.

#B The customer stores items in their Active and Saved lists

#C What it is

#D Who's selling it

#E How many they want

#F how many we have

#G what it currently costs

#H What we've said it costs

#I Everything we know about it (color, material, weight, etc.)

The design is a mix of patterns and styles that reflects its many hands. Some behavior was written in an object oriented(ish) style on the Cart object. For instance, here's the logic for adding new items.

Listing 9.2 The main logic for adding cart item

```
public class Cart {  
    List<CartItem> active;  
    List<CartItem> saved;  
    ...  
    private String qualifiedId(CartItem item) {      #A  
        return item.productId() + item.sellerId();    #A  
    }                                              #A  
    public void addOrUpdateItems(AddItemsRequest request) {  
        List<CartItem> items = this.convertToCartItems(request);      #B  
        var activeItemsById = toMap(this.active, this::qualifiedId); #C  
        var savedItemsById = toMap(this.saved, this::qualifiedId);  #C  
        for (CartItem incoming : items) {  
            String key = this.qualifiedId(incoming);  
            if (activeItemsById.containsKey(key))                #D  
                && !savedItemsById.containsKey(key)) {          #D  
                CartItem existing = activeItemsById.get(key);   #D  
                existing.setDesiredQty(                      #D  
                    existing.getDesiredQty() + incoming.getDesiredQty()) #D  
            );  
        }  
        else if (savedItemsById.containsKey(key)) {          #E  
            CartItem saved = savedItemsById.get(key);       #E  
            this.saved.remove(saved);                      #E  
            saved.setDesiredQty(                          #E  
                saved.getDesiredQty() + incoming.getDesiredQty()) #E  
            );  
            if (activeItemsById.containsKey(key)) {          #F  
                var activeItem = activeItemsById.get(key);    #F  
                activeItemsById.get(key).setDesiredQty(        #F  
                    activeItem.getDesiredQty() + saved.getDesiredQty()) #F  
                );  
            } else {                                         #G  
                this.active.add(saved);                     #G  
            }  
        }  
        else {  
            this.active.add(incoming);                  #H  
        }  
    }  
}  
private List<CartItem> convertToCartItems(AddItemsRequest request) {  
    return request.getItems().stream().map(external -> {  
        CartItem item = new CartItem();  
        item.setProductId(external.getProductId());  
        item.setSellerId(external.getSellerId());  
        item.setDesiredQty(external.getQuantity());  
        return item;  
    }).toList();  
}  
...  
public static <K,V> Map<K,V> toMap( #I  
    Collection<V> items,  
    Function<V,K> keyGetter) {  
    return items.stream().collect(  
        Collectors.toMap(keyGetter, Function.identity()));  
}
```

#A Important domain information! We need both a Product Id (what's being sold) and a Seller ID (who's selling it) to identify an item.

#B First we get the new items into roughly the right shape

#C To make lookups easier and more efficient, the lists are turned into Maps keyed by Product and Seller

#D Here's requirement 3.1. If anything overlaps with the active cart, we merge the items

#E Requirement 3.2 is far more complicated. It moves the item out of the Saved list and into the Active one

#F Special care is needed. The same item might be in both Active and Saved, so all quantities need merged together

#G if it's "just" in saved, we move the updated item to Active

#H Finally, an "easy" requirement (3.3): if it's not in either list, add it to Active. Although, looks are deceiving. There's a really subtle bug here that we'll cover later.

#I (We'll use this utility method throughout the chapter to help examples fit on the page)

The rest is piecemeal and scattered throughout the main controller. Take a few minutes to study this. It's meant to feel overwhelming. This is real code that was powering a wholesaler retail site (anonymized to protect the guilty, of course). The first time I encountered this it felt like showing up at a crime scene. It gets worse the longer you look at it.

Listing 9.3 Adding items to a cart. How hard could it be?

```
class AddItemsRequest {                                     #A
    @NotEmpty @Unique List<ItemInfo> items;
}
class ItemInfo {                                       #A
    @NotBlank @NotNull String productId;
    String sellerId;
    @Min(1) long quantity;
    BigDecimal displayedPrice;
}
class StoreController {                                #B
    ... // services elided

    @Post @Path("/add-items")
    public Response addCartItems(Session session, AddItemsRequest request) {
        Cart cart = cartRepo.loadCart(session.getCustomer()); #C
        var errors = cartItemValidator.validate(request);      #C
        if (!errors.isEmpty()) {                             #C
            return Response.BAD_INPUT(errors);             #C
        }
        cart.addOrUpdateItems(request);                      #D
        cart.getActive().forEach(item -> {                #E
            updateMetadata(item);                          #E
            updateInventory(item);                        #E
        });
        this.checkForPriceDriftAndUpdate(cart);           #E
        for (CartItem item : cart.getActive()) {          #E
            if (item.getDesiredQty() > item.getAvailableQty()) { #F
                if (item.getAvailableQty() > 0) {           #F
                    CartItem copy = item.copy();            #F
                    copy.setDesiredQuantity(               #F
                        item.getDesiredQty() - item.getAvailableQty()); #F
                    item.setQuantity(item.getQuantityAvailable()); #F
                    cart.getSaved().add(copy);              #F
                } else {                                #F
                    cart.getActive().remove(item);       #F
                    cart.getSaved().add(item);            #F
                }
            }
        }
        cart.recalculateSubtotal();                         #G
        this.checkIfEligibleForFreeShipping(cart);        #H
        this.updateRecommendedItems(cart);               #H
        cartRepo.save(cart);
        return Response.OK(cart);
    }
    private void updateInventory(CartItem item) {          #I
        var response = inventoryService.checkInventory(item.getProductId());
        item.setAvailableQty(response.getAvailableQty());
    }
    private void updateMetadata(CartItem item) {           #I
        String sellerId = item.getSellerId();
        Details details = item.getProductDetails();
        var response = productService.getProductDetails(item.getProductId());
        details.setCategory(response.getCategory());
        // Setting brand, color, material etc. Elided for space
        if (sellerId == null) {                           #J
            sellerId = response.getPromotedSeller();   #J
        }
    }
}
```

```

        item.setSellerId(sellerId);
    }

private void checkForPriceDriftAndUpdate(Cart cart) { #K
    for (CartItem item : cart.getActive()) {
        Offer offer = sellerService.getOffer(item.getSellerId());
        if (!offer.getCurrentPrice().equals(item.getLastNotifiedPrice())) {
            item.setLatestPrice(offer.getCurrentPrice());
            cart.setHasPriceChanged(true);
        }
    }
}

```

#A These are the main inputs to the system. The annotations tell us a bit about expectations, but not the whole story.

#B Dependencies are skipped to try to save space

#C The code is superficially reasonable. It starts off by vetting its inputs.

#D Then it starts its business logic (this was defined above in Listing 9.2)

#E As is often the case with many hands, a lot of logic is ad-hoc and written wherever needed

#F This is the code for requirement 5.0. It “balances” the cart to move unavailable items to the Saved list.

#G Computes various subtotals and sub-subtotals (like total items that count towards getting faster shipping)

#H The implementations for these are elided so we can focus on bigger crimes

#I Various helpers for various business logic.

#J For legacy reasons nobody understands or remembers, seller info isn’t always available when the request is made. In these cases, we fallback to the promoted seller.

#K Requirement 4.0 flags any items in the cart that have changed price.

That’s where we’re starting. It’s a mess, but one that can be fixed with the right piece of data.

9.2 Analyzing common problems in code

Finding flaws in existing programs is a great way to learn to write better ones. Problems need identified to be fixed, but few refactors start that way. Too often, we jump in and try to “fix” things by auto-piloting our way through a kind of architectural pattern matching bingo based on what we believe good code “should” look like. This refactoring usually happens at a whiteboard with a fat marker and completely disconnected from the details.

It’s easy to fall into the trap of trying to solve problems by encapsulating (hiding) them. If we were to make this mistake, it would probably start by asking object-y questions about the responsibilities of the Cart and its interfaces. Should a Cart Have-An Inventory? That feels backwards. Maybe we wrap the Cart in a CartService? That would hide the current problems and give us a place for dependencies. But, thinking more, users do more than just put things into a cart. They also checkout, process returns, track orders, and more. So, maybe we wrap the CartService in a bigger StoreService? Maybe there’s a design pattern here. Hmm...

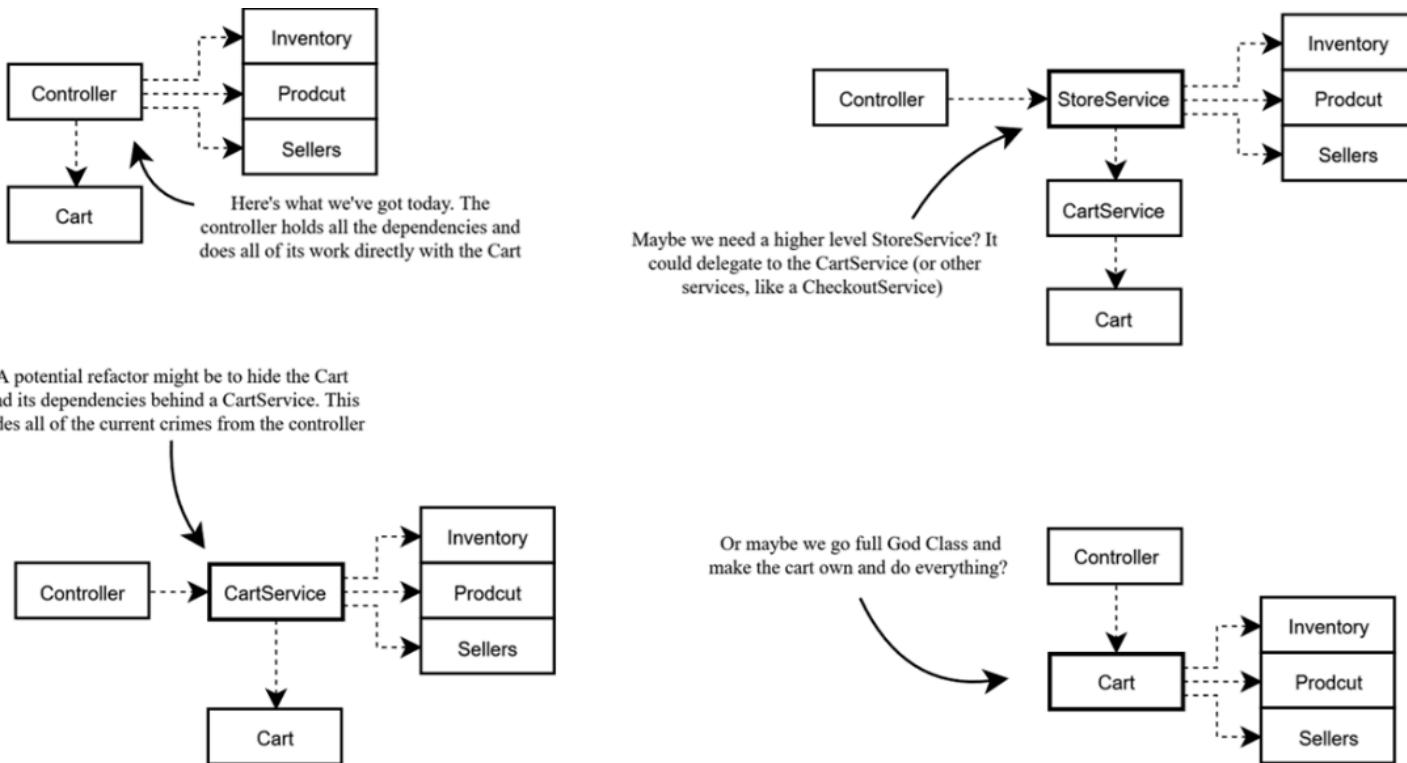


Figure 9.3 Boxes and arrows driven refactors for improving the current design

Drawing responsibility boundaries too early is an easy way to end up with a bunch of theoretically good architectural *stuff*, but none which solves the real problem.

Let's say we pick one of these. Now what?

Somewhere there will still be a Cart and a slew of crimes being committed against it. Indirection only moved the problems. We, the poor programmers working on the code, still have to don filthy coveralls, turn on our headlamps, and descend into the dark caverns where we've hidden the Cart.

So, let's not move anything around yet. Indirection won't solve our issues. The first step is just studying the code. Not every problem needs fixed, but every problem needs identified. Without that, refactors will only ever make the code different, not better.

9.2.1 Messy data modeling

One of the initial problems we might note with the Entities is the lack of cohesion in their modeling. Most of the Cart's representation has nothing to do with holding shopping items. It's mostly a dumping ground for ancillary concerns.

Listing 9.4 The Cart is mostly concerned with everything other than being a cart

```
public class Cart {  
    ...  
    List<CartItem> active; #A  
    List<CartItem> saved; #A  
    BigDecimal subtotal;  
    boolean hasPriceChanged; #B  
    List<CartItem> recommended; #B  
    boolean hasGiftOptions; #B  
    ... and so on #B  
}
```

#A Things related to being a Shopping Cart

#B Things not related to being a Shopping Cart

The items stored in the cart share the same problem. They're forced to play the role of an Inventory Tracker.

Listing 9.5 Items inside the cart track the inventory outside it

```
public static class CartItem {  
    @Id String id;  
    ...  
    long desiredQty;  
    long availableQty; #A  
    ...  
}
```

#A A snapshot of the inventory. Why this ended up here is lost to time (but likely summed up as "because it was convenient").

But the biggest problem with this modelling is what it allows the rest of the code to do. Rather than prevent illegal states, it propagates them.

Let's explore how.

9.2.2 Uncontrolled mutation

The Cart's internal state is directly and continuously mutated by code outside of it. Sometimes this mutation is done through the Cart's various behavioral methods, but most of the time external code reaches its fingers into the Cart and starts fiddling with knobs.

Listing 9.6 It's almost easier to count the lines that don't mutate something

```
cart.addOrUpdateItems(request); #A
cart.getActive().forEach(item -> {
    updateMetadata(item); #A
    updateInventory(item); #A
});
this.checkForPriceDriftAndUpdate(cart);
for (CartItem item : cart.getActive()) {
    if (item.getDesiredQty() > item.getAvailableQty()) {
        if (item.getAvailableQty() > 0) {
            CartItem copy = item.copy();
            copy.setDesiredQuantity(
                item.getDesiredQty() - item.getAvailableQty());
            item.setQuantity(item.getQuantityAvailable()); #A
            cart.getSaved().add(copy); #A
        } else {
            cart.getActive().remove(item); #A
            cart.getSaved().add(item); #A
        }
    }
}
cart.recalculateSubtotal(); #A
this.checkIfEligibleForFreeShipping(cart); #A
this.updateRecommendedItems(cart); #A
cartRepo.save(cart);
return Response.OK(cart);
```

#A All modify the internal state of the Cart

A lot of this mutation happens in the shadows, several layers down the call stack.

Listing 9.7 Mutating a reference three layers removed is especially egregious

```
private void updateMetadata(CartItem item) {
    String sellerId = item.getSellerId();
    Details details = item.getProductDetails();
    var response = productService.getProductDetails(item.getProductId());
    details.setCategory(response.getCategory()); #A
    // other setters skipped for space
    if (sellerId == null) {
        sellerId = response.getPromotedSeller();
    }
    item.setSellerId(sellerId);
}
```

#A This is a method in the StoreController calling a setter on the Details class, which is on a CartItem, which is inside of a list, which is inside of the Cart. It's so excessive that it starts to feel rude.

Rampant mutation like this is a common problem with mutable Entities and ORMs. Even if Entities weren't just thin veneers over the underlying database representation (as most are), it would still be really hard for even the most principled, careful designer to maintain encapsulation boundaries as more and more demands are placed on an Entity. Many problems don't fit into the verb attached to a noun model of objects. Ownership gets too weird. Sometimes all you really want is the data. So, people start reaching directly inside of the object to touch its state.

It's hard to claw back control from an object that has lost its encapsulation. You can try defensive programming, but human memory is faulty and institutional knowledge is

fleeting. Every mutable reference creates a potential failure point. Invariants applied only sometimes are, by definition, not invariants.

Listing 9.8 Remembering to check invariants works until it doesn't

```
class Cart {  
    void addOrUpdateItems(AddItemsRequest request) {  
        for (CartItem incoming : items) {  
            // all the merge logic here  
        }  
        this.assertInvariants() #A  
    }  
    private void assertInvariants() {  
        // make sure we didn't end up with duplicates, and that  
        // each item is in the right cart (active vs saved) based  
        // on available inventory and desired quantities  
    }  
  
    public List<CartItem> getActive() {  
        return this.active; #B  
    }  
}  
"  
void someActionSomewhereElse(Cart cart) { #B  
    ...  
    cart.getActive().getFirst().setDesiredQty(10); #B  
}
```

#A We can try to defend and verify the object.

#B But as long as we're handing out mutable references, our defenses can be bypassed.

Uncontrolled mutation is a problem, but it's not necessarily *the* problem. Tons of objects mutate their internal state without causing any issues. There's something wrong deeper in the code's design.

Let's keep digging.

9.2.3 Sequential coupling

Sequential coupling (sometimes called "temporal coupling") is when the methods exposed on an object have to be called in a specific sequence in order to do the right thing. One of the most basic sells of encapsulation is that we shouldn't have to know what's behind an interface to use an object. Sequential coupling violates this principle and puts the burden on user of the object to just "know" that some methods can only be called before or after others without breaking something.

Listing 9.9 Sequential coupling, or how to break everything by moving anything

```
cart.addOrUpdateItems(request);                      #A
cart.getActive().forEach(item -> {
    updateMetadata(item);                           #A
    updateInventory(item);                         #A
});;                                              #A
this.checkForPriceDriftAndUpdate(cart);             #A
-----
// Changing the order changes the program
cart.addOrUpdateItems(request);
→ this.checkForPriceDriftAndUpdate(cart);           #B
cart.getActive().forEach(item -> {
    updateMetadata(item);
    updateInventory(item);
});
this.checkForPriceDriftAndUpdate(cart);
```

#A In the original program we add the items, then refresh the metadata and inventory state, then check for price drift. Each line secretly depends on the one before it in order to produce a valid answer.

#B Reordering even a single line leaves the cart in an invalid state

It's easy to accidentally introduce sequential coupling into an object. Look for it during code review and you will find it. It often happens as a side-effect of trying to apply other "good" design principles. Single responsibility is the usual culprit. When taken too far (or misunderstood, as it often is) Single Responsibility can nudge us towards writing methods that are atomic from the perspective of feature, but non-atomic from the perspective of the object.

Methods like `updateMetadata` and `updateInventory` are about one "responsibility." They're superficially well named and well scoped, but their single-ness hides an incompatibility with the design of the object they're modifying. Despite each doing one thing, the actions aren't atomic. The cart is left in a state of turmoil until other methods are called *later*.

Listing 9.10 Each is "about" one thing, but each leaves the cart in an inconsistent state

```
cart.addOrUpdateItems(request);          // Invalid      #A
cart.getActive().forEach(item -> {
    updateMetadata(item);              // Invalid      #B
    updateInventory(item);            // Invalid      #B
});;                                  // Invalid      #B
this.checkForPriceDriftAndUpdate(cart); // Invalid      #C
for (CartItem item : cart.getActive()) { // Invalid      #D
    // cart balancing logic        // Invalid      #D
}
cart.recalculateSubtotal();            // Invalid!     #D
...
// Valid!                            #E
```

#A Invalid, because items are missing details and might be in the wrong list

#B Invalid, because new inventory counts might require changing Active items to Saved

#C Invalid, this updates metadata, but leaves Cart Items in their potentially incorrect list

#D Invalid, we've balanced the cart, but not the totals

#E Valid (finally!). The state of the cart is brought into alignment with the state of its items.

Sequential coupling in this code is a big problem, but its existence is usually a symptom of a still deeper problem in the object's design. Let's keep going!

9.2.4 Hidden knowledge, partial construction, and shotgun validation

Behind sequential coupling is an object that's either incomplete, invalid, or in an undefined state of partial construction. In our example code, the problems begin the moment we hand the request over to the `addOrUpdateItems` method.

Listing 9.11 Where most of our problems begin

```
class Cart {  
    private List<CartItem> active;  
    ...  
    public void addOrUpdateItems(AddItemsRequest request) {  
        List<CartItem> items = this.convertToCartItems(request); #A  
        var activeItemsById = indexBy(this.active, this::qualifiedId)  
        var savedItemsById = indexBy(this.saved, this::qualifiedId)  
        for (CartItem incoming : this.active) {  
            // logic elided  
        }  
        ...  
    }  
    private List<CartItem> convertToCartItems(AddItemsRequest request) {  
        return request.items().stream().map(external -> {  
            CartItem item = new CartItem(); #B  
            item.setProductId(external.getProductId()); #C  
            item.setSellerId(external.getSellerId()); #C  
            item.setQuantity(external.getQuantity()); #C  
            return item; #C  
        }).toList();  
    }  
}
```

#A The original sin! This converts the request items into a familiar shape, but only partially

#B An empty constructor means we get whatever we get. Maybe some fields might get defaults. Most will probably be null. Who knows.

#C Only the fields we care about right now get set. Everything is left in an unknown or undefined state until it gets set later

The code wants everything to be in the same shape so it can move items between lists, so the first thing it does is try to convert the incoming items into a `CartItem`. But it can't do it completely. The incoming request doesn't have enough information. Instead, the code constructs it *partially* with just the pieces available on the request.

Partial construction is a common problem in Entity heavy code. A single representation cannot fit every need in an application. Different contexts will have different information available. The input to this method only knows a little about the world, but it's forced to stretch until it fits the shape needed by the entity. And the only way to do that is by setting the fields it can while leaving everything else blank.

The most damaging thing a codebase can do is hide what it knows. This code knows about secret behaviors that aren't stated in the requirements or validated at the front door. The hidden knowledge leads to a remarkably subtle bug that can cause duplicates to end up in the cart.

Listing 9.12 Dangerous hidden knowledge

```
class AddItemsRequest {  
    @NotEmpty List<ItemInfo> items;  
}  
class ItemInfo {  
    @NotBlank @NotNull String productId;  
    String sellerId;      #A  
    @Min(1) long quantity;  
    BigDecimal displayedPrice;  
}  
class StoreController {  
    ...  
  
    public Response addCartItems(Session session, AddItemsRequest request) {  
        Cart cart = cartRepo.loadCart(session.getCustomer());  
        Set<ValidationException> errors = cartItemValidator.validate(request); #B  
        if (!errors.isEmpty()) {  
            return Response.BAD_INPUT(errors);  
        }  
        cart.addOrUpdateItems(request);      #C  
        cart.getActive().forEach(item -> { #C  
            updateMetadata(item);    #C  
            updateInventory(item);  
        });  
        this.checkForPriceDriftAndUpdate(cart);  
        ...  
        return Response.OK(cart);  
    }  
    ...  
    private void updateMetadata(CartItem item) {  
        String sellerId = item.getSellerId();  
        ...  
        if (sellerId == null) {           #D  
            sellerId = response.getPromotedSeller(); #D  
        }  
        item.setSellerId(sellerId);  
    }  
}
```

#A The original developer didn't call this out as nullable or Optional, but we find out later in the code that it can be!

#B This silently passes through validation. We can't validate what we don't know should be validated.

#C That null is propagated through the code

#D It's finally detected and set here. It feels unfair! This isn't in the requirements! What the heck!

The critical thing to notice is that `sellerId` is set *after* we've already passed through validation and *after* we've started processing this data.

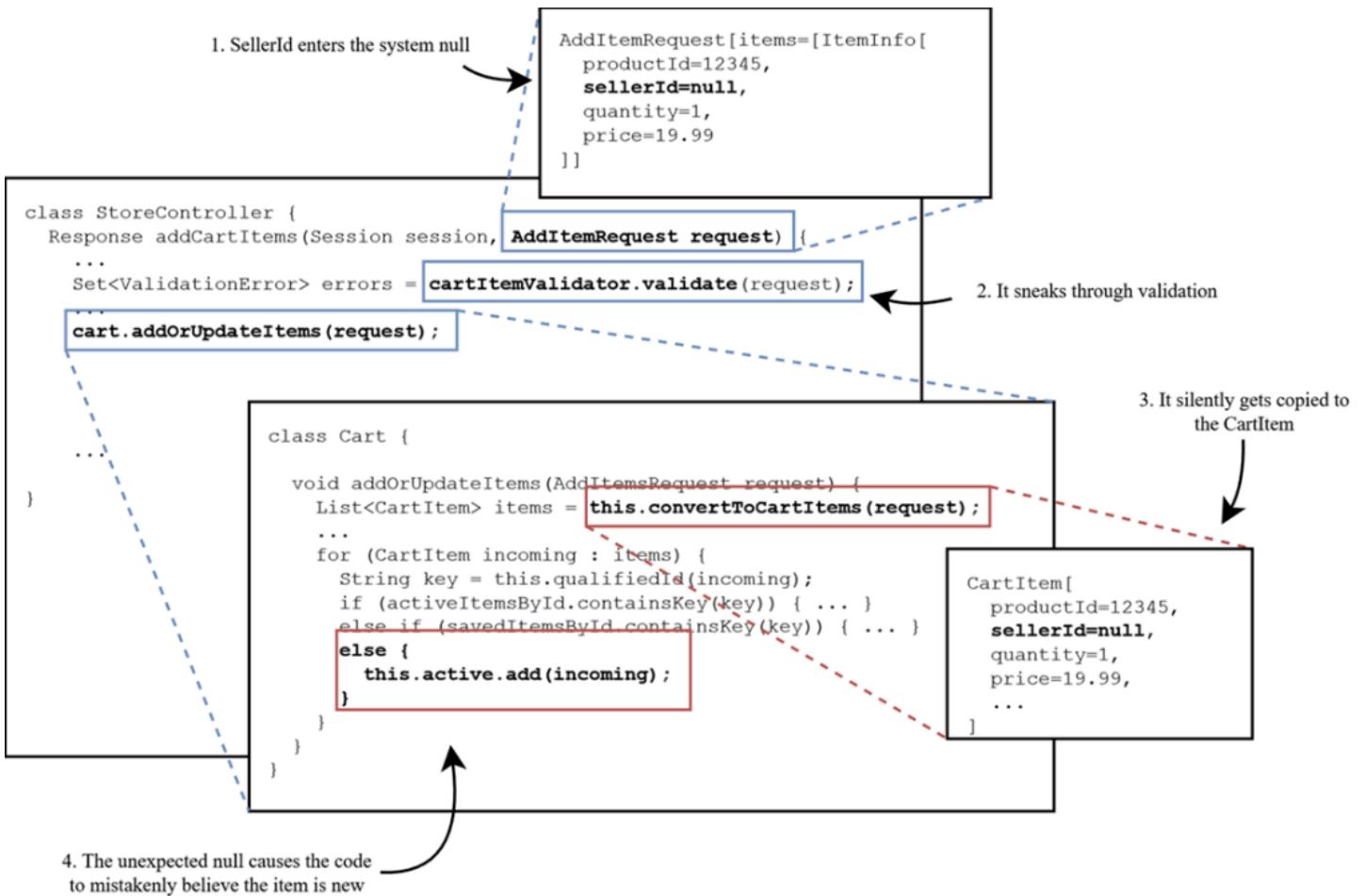


Figure 9.4 How a single null can sneak through a system

By the time sellerId is finally set it's too late. We've already unknowingly put our system into an invalid state.

Listing 9.13 Nulls cause us to silently propagate invalid states

```

public class Cart {
    List<CartItem> active;
    List<CartItem> saved;
    ...
    private String qualifiedId(CartItem item) {
        return item.getProductId() + item.getSellerId(); #A
    }
    public void addOrUpdateItems(AddItemsRequest request) {
        List<CartItem> items = this.convertToCartItems(request);
        var activeItemsById = indexBy(this.active, this::qualifiedId)
        var savedItemsById = indexBy(this.saved, this::qualifiedId)
        for (CartItem incoming : items) {
            String key = this.qualifiedId(incoming); #B
            if (activeItemsById.containsKey(key)) {...} #B
            else if (savedItemsById.containsKey(key)) {...} #B
            else {
                this.active.add(incoming); #C
            }
        }
    }
}

```

#A This doesn't throw a Null Pointer when sellerId is null, it silently appends the string "null"

#B So, all subsequent lookups will fail

#C And because nothing matched, we'll add this to the end of the list on the mistaken assumption that it's a new item

Every defense is bypassed when the Cart Item is updated. What was previously null gets an ID assigned, and (given an unlucky roll of the dice) ends up duplicating something already in the Customers cart. What happens next is undefined. Double charged? Under charged? Who knows!

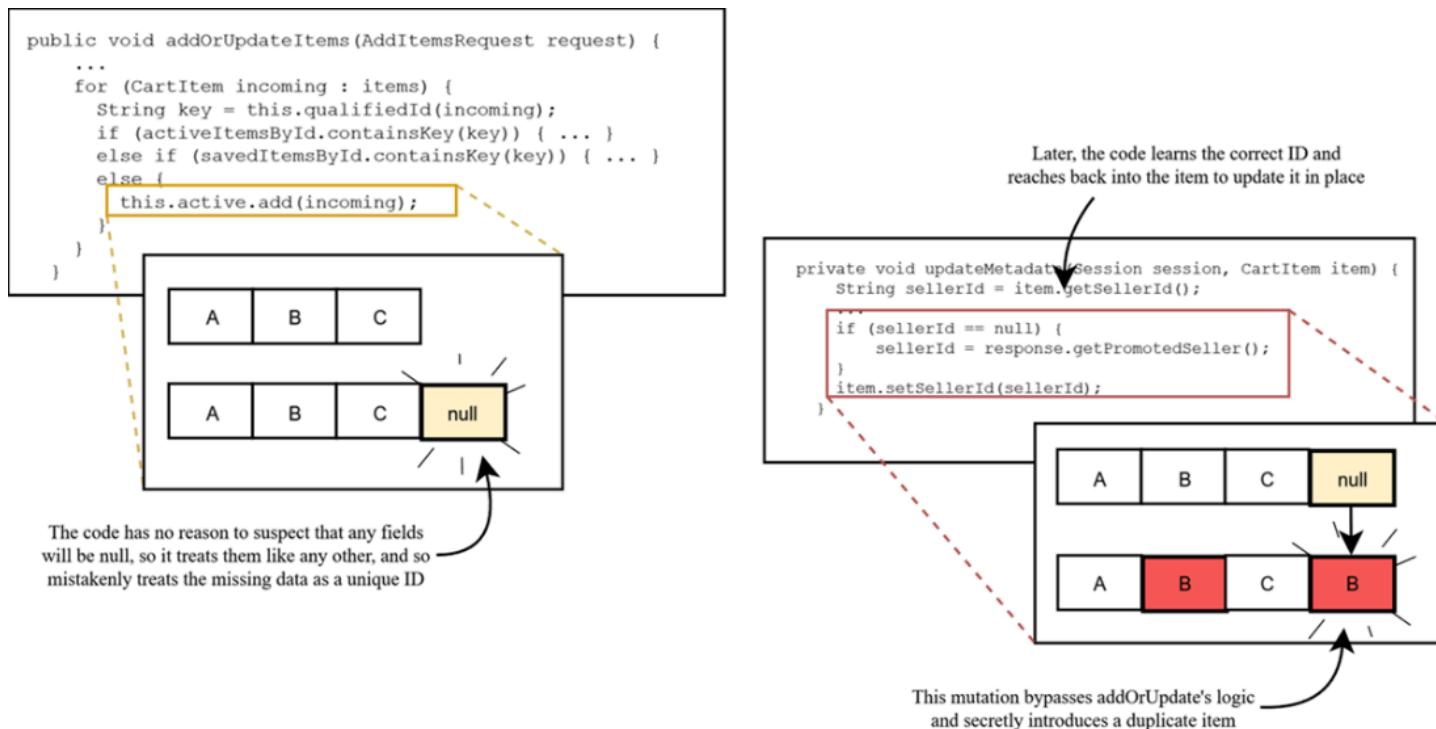


Figure 9.5 This took me hours careful reading to uncover

What makes these types of problems insidious is that the individual methods are “fine.” It’s only when combined in a particular sequence, and with a particular set of values, that the bug rears its head. Everything works until it doesn’t.

Programming with objects that haven’t been fully constructed or validated seems preposterous when pointed out, but hindsight bias makes all problems appear “obvious.” It’s easy to do accidentally. It’s made even easier when many hands are touching the same code paths. Mutation contributes, but isn’t the sole cause. Even the most principled codebases encounter this problem. All it takes is a shift in requirements or semantics to create a path through your code where invariants are *assumed* rather than enforced. The problem is so prevalent that security researchers have summed it up with the delightfully fun name: Shotgun Parsing.

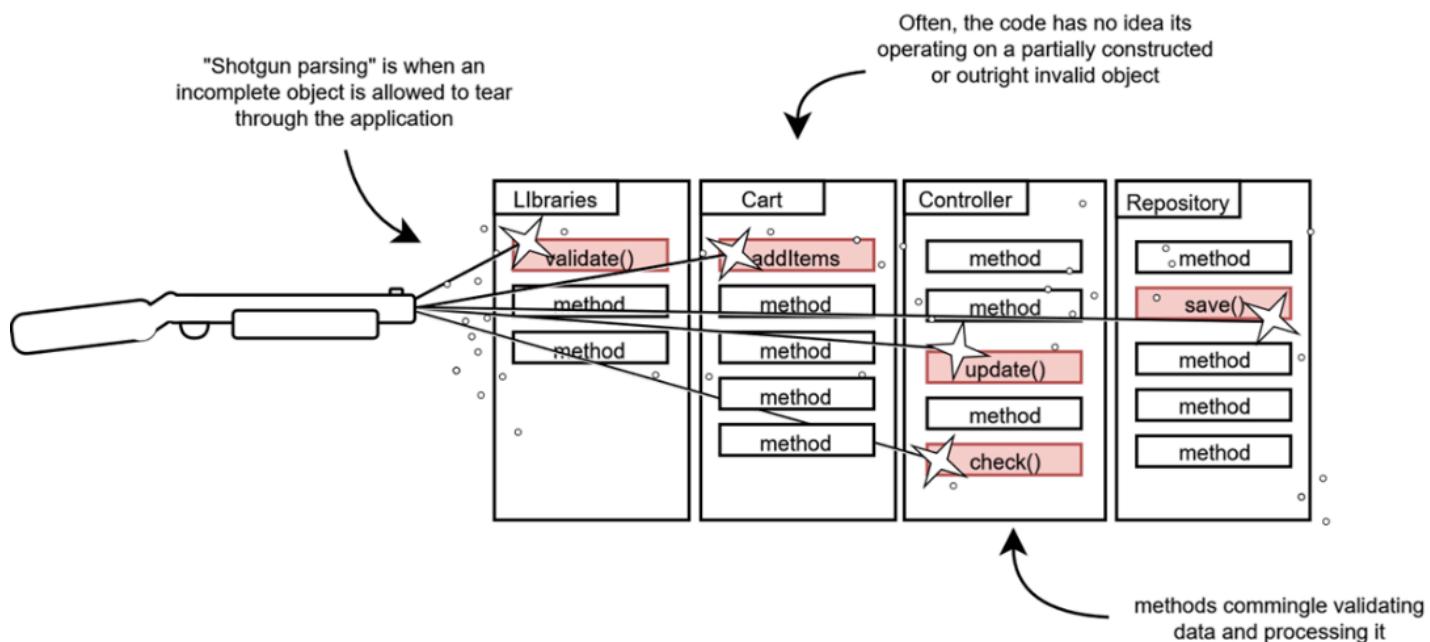


Figure 9.6 Shotgun parsing

If the code in listing 9.3 doesn’t feel like a shotgun going off, I don’t know what does. It’s filled with lies, assumptions, hidden knowledge, and secret functionality that’s not stated in the requirements. These gaps tear through the code leaving destruction in their wake.

9.3 Fixing the problems

A good refactor fixes problems in the code. A great refactor prevents them from reoccurring. People are drawn to the patterns they find in an existing codebase. Now is the time to use our modeling powers to make illegal states impossible to represent. Our refactor is successful when the people who come after us fall into pits of success, rather than despair.

Where do we start?

With data!

9.3.1 Work backwards from invariants

Let's refresh ourselves on what we know. Here are the requirements from section 9.1

Table 9.2 a refresher on the requirements

3	When a customer adds items to their cart
3.1	If the item is already in the Active cart, the service shall add their quantities
3.2	If the item is already in the Saved cart, the service shall add their quantities and move the item to Active
3.3	If the item is not in either cart, it shall be added to Active
5.0	The system shall automatically move unavailable Active items to Saved

The first thing we should do is formalize what these requirements mean. Encoding them as data prevents the invalid states which plagued the original design. To do that, we have to ask our question from chapter 7: *what does it mean to be correct?* How do we translate the requirements into laws that hold “for all states.”

Developing this “for all” muscle takes time and effort, so it’s worth practicing every chance we get. English prose is frustratingly imprecise. Even the most “formal” business requirements require refinement.

One way to tackle this is to ignore what the requirements tell us to *do* and instead focus on what their end result *is*. This is tricky, because most requirements describe logic (what it should do), not state (what it should be). We have to look past the verbs and find the nouns. When all the actions are done, what must be true about a Cart “for all states” it could possibly enter?

Requirement 5.0 gives us a foothold. It’s phrased as an action, but we can pick apart its prose to formalize what it’s saying about the Cart’s state. For instance, “Unavailable” is something we can define. It means that regardless of how items move around, the desired quantity should never exceed the quantity available in the inventory. Or, said formally, $\text{Desired} \leq \text{Available}$ for all items in the Active cart.

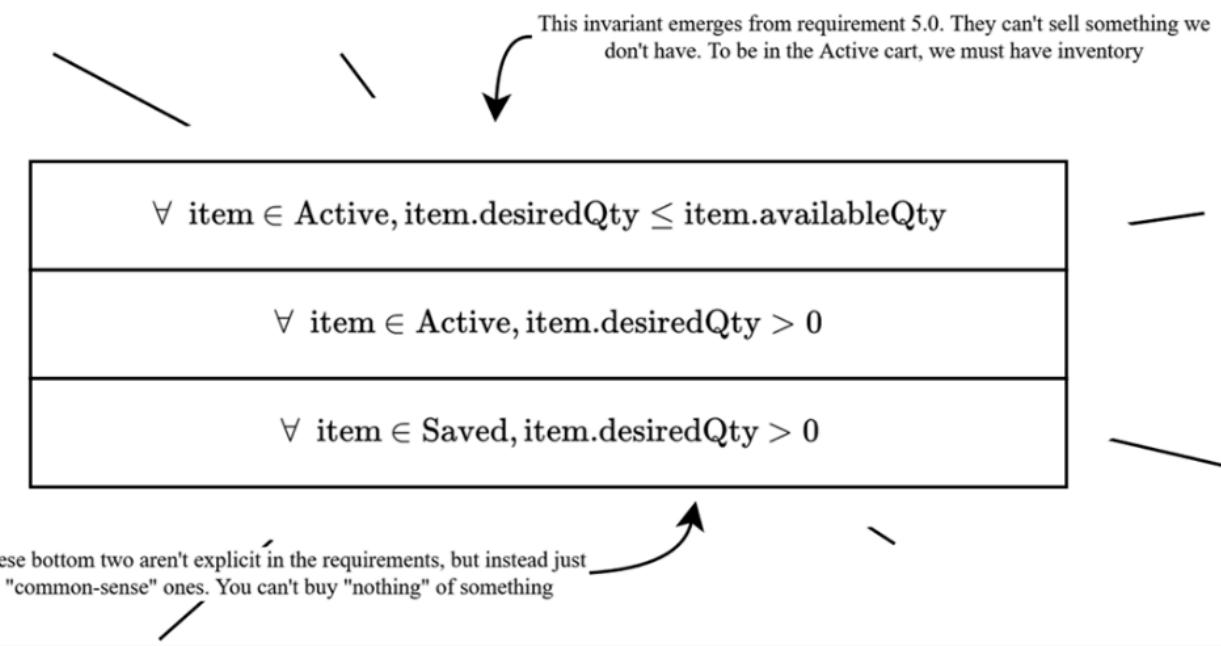


Figure 9.7 Expressing the Cart’s invariants formally

We can work backwards from these invariants to sketch a new version of the Cart. Unlike objects, which need their internal state continuously guarded prevent corruption, immutable data has exactly one unchanging state. We only need to defend its invariants during construction. If it's valid when we create it, it's valid for eternity. Immutable data cannot be corrupted.

Let's try to model this.

Listing 9.14 A first stab at the modeling

```
record ValidCart(      #A
    List<Item> active,
    List<Item> saved
) {
    ValidCart {
        for (Item item : active) { #B
            if (item.desiredQty() >= item.availableQty()
                || item.desiredQty() <= 0) {
                throw new IllegalArgumentException("...");
            }
        }
        for (Item item : saved) { #B
            if (!(item.desiredQty() > 0)) {
                throw new IllegalArgumentException("...");
            }
        }
        active = List.copyOf(active); #C
        saved = List.copyOf(saved);   #C
    }
}
```

#A No mistaking what we mean. The name and its invariants encode the semantics we desire.

#B (stream().allMatch() would be good here, but it's hard to fit on a book page)

#C This data model is a representation of what we know, but it's also a defense against the ingrained patterns in the codebase. Creating a defensive Unmodifiable copy on the way in prevents old habits from wreaking havoc.

This is a solid first stab. It lines up with the general mental model of a Cart. It enforces invariants. It also ensures immutability by creating defensive copies. There's lots to like.

But it could be better.

There's still stuff we know about the cart that's not captured in the modeling. Namely, that every item should be unique. Storing items in a List betrays that knowledge.

Maybe a Set would be better?

Listing 9.15 A second attempt

```
record ValidCart(
    Set<Item> active,    #A
    Set<Item> saved     #A
) {
    ValidCart {...}
}
```

#A Swapping from List to Set

This is closer in spirit but still doesn't correctly capture the semantics. Uniqueness in Java is based on the *entire* object (unless you do some nasty hacking). Two cart items with the same IDs would still be treated as different if they varied in any other attribute (say, price). They'd be unique from the perspective of the Set but duplicated from the perspective of our feature.

Maybe a few more defenses in the constructor?

Listing 9.16 Recreating the original problems in a new spot

```
record ValidCart(  
    Set<Item> active,  
    Set<Item> saved  
) {  
    ValidCart {  
        Set<String> byId = active.stream()  
            .map(x -> x.sellerId + x.productId) #A  
            .collect(Collectors.toSet());  
  
        if (active.size() != byId.size()) {  
            throw ...  
        }  
        // other invariants skipped for brevity  
    }  
}
```

#A What uniqueness means is completely hidden away from the code except for this one line in the constructor

But that's a step backwards in our modeling. This is why it's always good to start with a deep analysis. One of the biggest problems in the original was just how much important knowledge was hidden. If we did this, we would be recreating the sin that plagued the original code. Refactors need to fix problems, not just move them!

How do we model uniqueness based on a subnet of attributes?

Maps!

This lets us encode our knowledge about what uniqueness means directly into our representation.

Listing 9.17 Modeling the Cart items as a Map

```
record ProductSeller(String productId, String sellerId){} #A

record ValidCart(
    Map<ProductSeller, Item> active,    #B
    Map<ProductSeller, Item> saved       #B
) {
    ValidCart {
        #C
        for (Item item : active.values()) {      #D
            if (item.desiredQty() >= item.availableQty()
                || item.desiredQty() <= 0) {
                throw new IllegalArgumentException("...");
            }
        }
        for (Item item : saved.values()) {          #D
            if (item.desiredQty() <= 0) {
                throw new IllegalArgumentException("...");
            }
        }
    }
}
```

#A What we know goes in the code! Cart items are naturally keyed by the combination of Product and Seller ID.

#B Swapping List and Set for Map

#C Note that we no longer need to assert uniqueness in the constructor. The data structure enforces it for us

#D The rest of the code is the same, just tailored for Maps.

I suspect this is at odds with your *presentational* mental model of a Cart. Customers interact with their cart as a list of items, not some weird map of items. But that mismatch is a feature! Our internal modeling of the cart doesn't need to match how we present it to the outside world or store it in a database. That was another mistake in the original code – it let multiple concerns tug its representation in conflicting directions and lost its ability to maintain invariants as a result.

It's worth pausing here to notice how little we've done, yet how much we've accomplished. So far, our "refactoring" is the introduction of a few data types. But they hold everything we know about the domain. Everything we learned during our analysis. Nothing is hidden. Everything is on display. Invariants defended.

As a side note, this milestone is a great time in the refactoring process to cut your first code review. Show your team. Do they understand it? Does it communicate what you think it does? Did we miss anything that someone else knows? Iteration at this point is as fast as it gets. We're not tied to any code. Data models can be created, revised, or thrown away entirely without pain.

Let's keep going.

Since we've placed the invariants on the Cart, the design for individual Items should be relatively simple. However, it requires an important design decision: what do we do with the inventory coupling from the original design?

As a reminder, it looks like this:

Listing 9.18 For some reason, Items in the Cart know about inventory statistics

```
public static class CartItem {  
    ...  
    long desiredQty;  
    long availableQty; #A  
    ...  
}
```

#A This really doesn't belong here.

The obvious and “correct” answer is to get those inventory counts out of there. Keeping them would destroy the cohesion of the data model we’re trying to create. We’d be falling into the very trap that we just tried to avoid. Problems would be moved, rather than solved.

And yet, I’m going to go against every idea in the book so far and suggest we keep this clearly incorrect coupling. At least for our *first* refactor.

My rationale is twofold. The first is that human dynamics are complicated. If a team is used to having those two fields available together, then “taking it away from them” can be poorly received. Extending an olive branch of familiarity in a refactor that’s changing paradigms can go a long way towards smoothing over objections based in emotion or arguments for convenience.

The second is to just show that it’s OK to give up some purity and do the “wrong” thing from time to time *as long as we’re able to defend our invariants*. Our current design is strong enough to absorb this quirk while still maintaining correctness. With Data, we construct it once, construct it right, and never worry about it again.

Here’s the how we’ll do it:

Listing 9.19 Modeling a cart item

```
record Item(  
    ProductSeller productSeller,  
    Details details,  
    long desiredQty,  
    long availableQty, #A  
    BigDecimal lastNotifiedPrice,  
    BigDecimal currentPrice #B  
){}  
}
```

#A A minor blemish on an otherwise cohesive and focused model

#B you could argue that this knowledge similarly belongs “outside” of the item, but we’ll keep it here for the same reasons (at least for now!)

One question might be whether this model is “enough.” Once you get the hang of using the type system to encode business rules, it can be alluring to flex your powers everywhere. For instance, we know that items in the Active list have special rules which apply to their quantities, whereas items in the Saved list do not. That’s important domain information! Should we encode it into the type system?

Listing 9.20 Bringing out the big type safety guns

```
sealed interface OrderQuantity {  
    long desired()  
    long available();  
  
    record Constrained(long desired, long available)      #A  
        implements OrderQuantity {  
            Constrained {  
                if (desired > available) { throw ... }  
            }  
        }  
    Record Unconstrained(long desired, long available)  #A  
        implements OrderQuantity {}  
}  
// then we could do something like...  
record Item<Quantity extends OrderQuantity> (  #B  
    ProductSeller productSeller,  
    Quantity qty, #B  
    ...  
){}  
  
// and finally something like...  
record Cart(  
    List<Item<Constrained>> active,  
    List<Item<Unconstrained>> saved  
){}
```

#A We could create a sealed family of Quantity types that have different constraints
#B And then parameterize our Items by the type of Quantity they should be carrying.

It's kind of neat, but I'd argue that it moves the invariants somewhere other than where they belong. An individual item has no real opinion about its attributes. It's only when put inside of a specific collection in the cart that we start to care about the relationship between its various quantities. The Cart is what determines the invariants. The items are just along for the ride.

PREVENT OLD HABITS FROM WREAKING HAVOC

Habits are slower to change than code. You have to help people fall into the pit of success. While a mature, philosophically aligned team can avoid nulls solely by convention, most teams with mixed experience levels and backgrounds cannot. New programming patterns take time. Instead of relying on convention, we can be explicit during this transitional period and make it absolutely clear that nulls are not allowed. The original design might have allowed SellerID to be null (until it was set somewhere), but sellerID in *our* model is *an invariant of the Cart*. It must be present.

Listing 9.21 Clarifying the expectations of the new world

```
record ProductSeller(  
    String productId,  
    String sellerId  
) {  
    ProductSeller {  
        Objects.requireNonNull(productId); #A  
        Objects.requireNonNull(sellerId); #A  
    }  
}
```

#A Prevents any old habits from poisoning our new data model

We could similarly do this everywhere that nulls might sneak in due to old habits.

Listing 9.22 Being really, really explicit about our expectations

```
record Item(  
    ProductSeller productSeller,  
    Details details,  
    long desiredQty,  
    long availableQty,  
    BigDecimal lastNotifiedPrice,  
    BigDecimal currentPrice  
) {  
    Item {  
        Objects.requireNonNull(id);  
        Objects.requireNonNull(details);  
        Objects.requireNonNull(desiredQty);  
        Objects.requireNonNull(availableQty);  
        Objects.requireNonNull(lastNotifiedPrice);  
        Objects.requireNonNull(currentPrice);  
    }  
}
```

It's a little tedious, but it goes a long way towards setting expectations and preventing errors.

While we're here, now is also the time to start properly encoding what *is* allowed to be optional. Much of the pain in the original implementation stems from hidden nulls. It's what caused us to unknowingly shotgun parse and operate on invalid data. The refactored system can be "hardened" to encode what we know (even while keeping the original mutable types).

Listing 9.23 Refactoring towards what we know

```
class AddItemsRequest {  
    @NotEmpty @Unique List<ItemInfo> items;  
}  
class ItemInfo {  
    @NotBlank @NotNull String productId;  
    String sellerId;  
    Optional<String> sellerId; #A  
    @Min(1) long quantity;  
    BigDecimal displayedPrice;  
}
```

#A Encoding what we know about optionality into the type system. No more hidden nulls!

9.3.2 Draw boundaries with data

We've spent a lot of time in this book exploring how modeling techniques can enrich our data types with meaning and semantics. However, data can do something even more powerful. It can create *boundaries*. These boundaries are special because we can place them anywhere. They don't need to be attached to an object or interface. They can be dropped into arbitrary points in our code. When we do this, data types take on a special role. They act as a gate keeper. They demand *proof*. If you want to access this code, you have to prove you can instantiate the data type it requires.

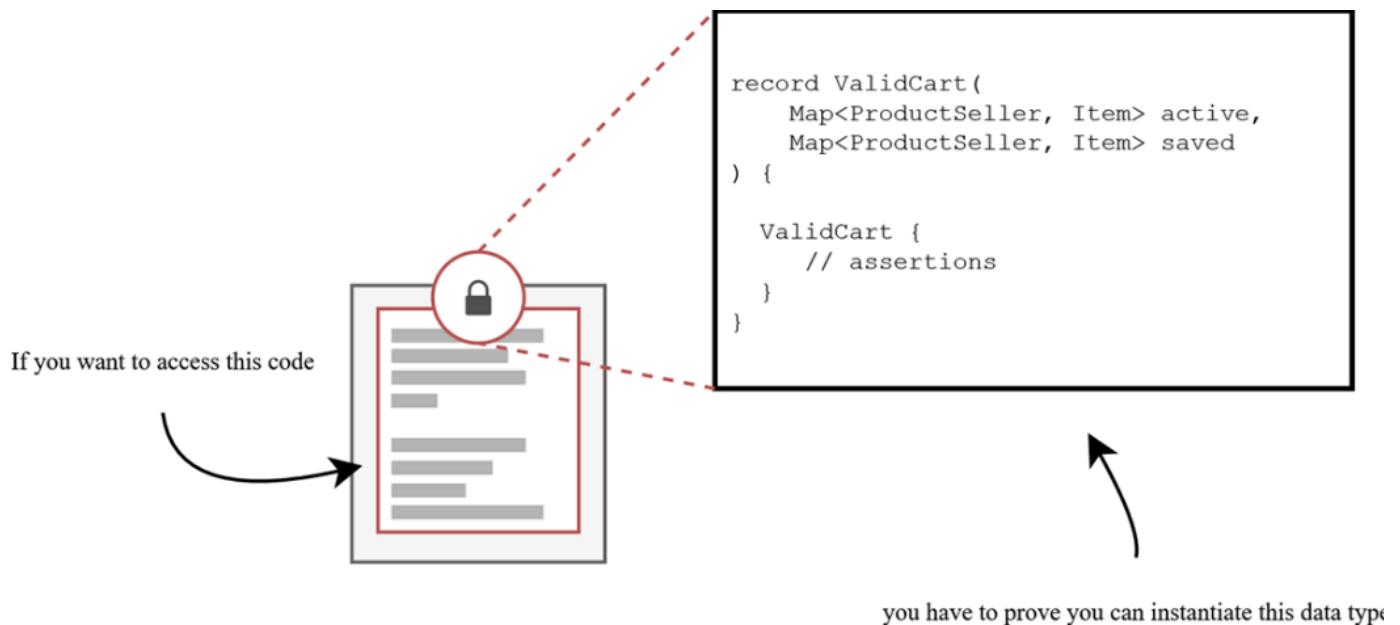


Figure 9.8 Protecting access to code with data

This thinking is the basis of something called the Curry-Howard Correspondence. Data Types are assertions. *Instances* of those data types are proof we can satisfy them. This idea initially feels weirdly circular and “obvious.” The only way to create a data type is, obviously, to have data that satisfies the demands of the type. But that obviousness is exactly the point. Data is more than a representation, it’s a demand.

Listing 9.24 Prove to me that you have what I need

```
public void checkout(ValidCart cart) {  
    // logic that's guarded by its input type #A  
}
```

#A If you want to run any of this code, you have to first prove that you can construct a piece of data that meets all of its demands.

The ability to drop a boundary anywhere removes a lot of the stress we face when designing boundaries around an interface. It's easy to tie yourself into a knot worrying if things are in the right layer, or using the right pattern, or if responsibilities are “singular” enough, or cohesive enough, or whatever other architectural concerns we weigh when establishing boundaries.

Instead, we get to ignore what things *do* and instead work backwards from what invariants they require. These boundaries can be as small as a few lines in a method, or

so big that they carve seams across the codebase.

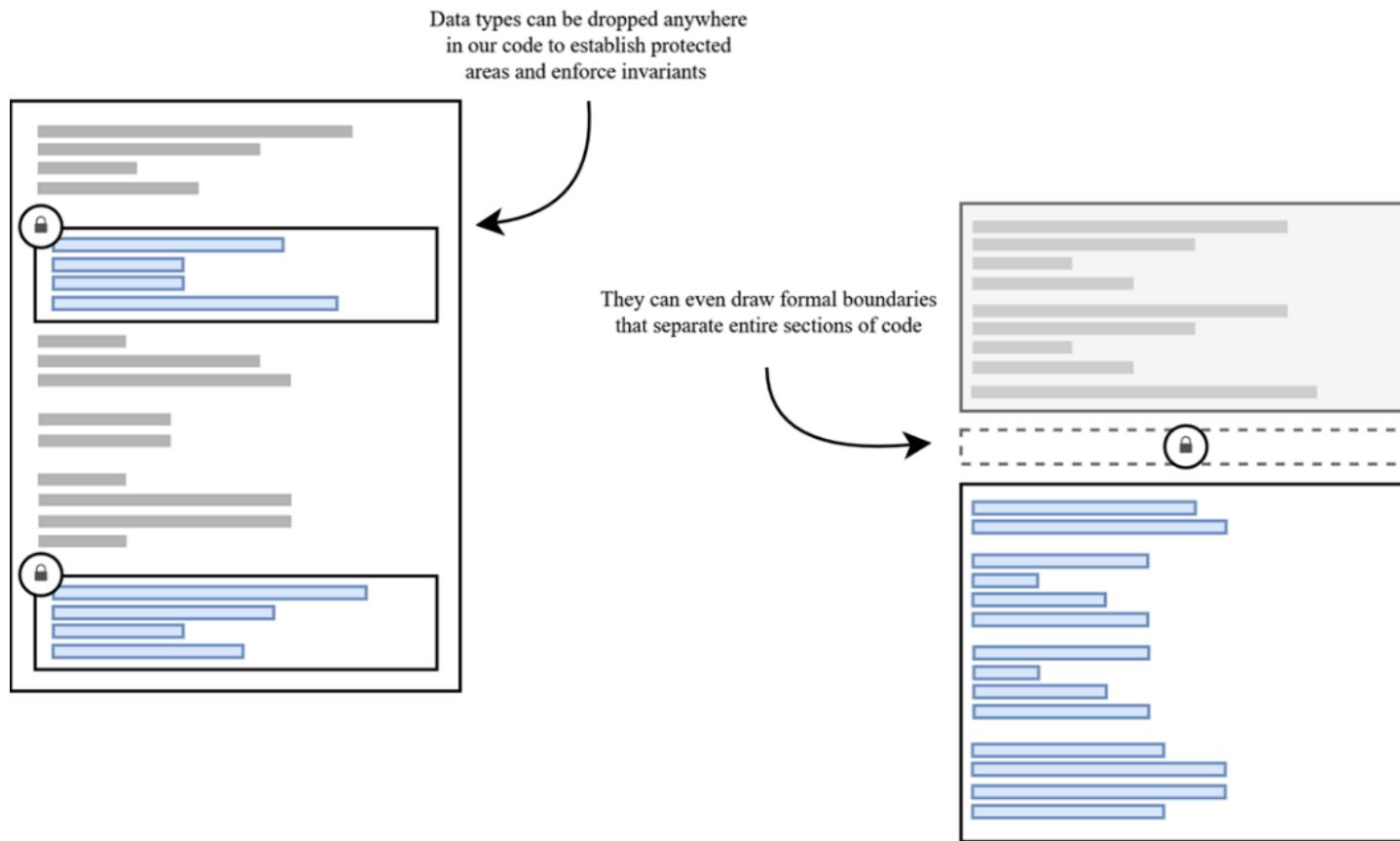


Figure 9.9 Drawing boundaries with data

Mentally, I like to imagine refactoring towards these boundaries as planting a flag in foreign territory. We're staking a claim. This part of the code now belongs to the new world. It's doing things different. And like all good imperialists, the goal is to grow our new sphere of influence.

Looking at our code, we would probably try to establish control somewhere *after* the items have been added to the cart, but *before* we start doing any other business logic. It's here that we can use our `ValidCart` data type to place a demand on the code. "You must be able to construct this data in order to continue."

Listing 9.25 Establishing a zone of control in foreign territory

```
@Post @Path("/add-items")
public Response addCartItems(...) {
    Cart cart = cartRepo.loadCart(session.getCustomer());
    var errors = cartItemValidator.validate(request);
    if (!errors.isEmpty()) {...}
    cart.addOrUpdateItems(convertToCartItems(request));
    cart.getActive().forEach(item -> {...});
    this.checkForPriceDriftAndUpdate(cart);
}

-----+
        BOUNDARY

ValidCart validCart = new ValidCart(...);      #A

-----+
cart.recalculateSubtotal();
this.checkIfEligibleForFreeShipping(cart);
this.updateRecommendedItems(cart);
cartRepo.save(cart);
return Response.render(cart)
}
```

#A We're making a demand right here in the middle of the code. In order to go any further, you have to be able to prove that you can create this data type and satisfy all its invariants.

This type separates the old and new worlds. Now we let it guide the refactoring.

Listing 9.26 Incrementally making code participate in invariants

```
List<Item> getRecommendedItems(ValidCart cart) {
    // logic for retrieving relevant recommended items  #A
}
Subtotals computeSubtotals(ValidCart cart) {
    // logic for computing totals                      #A
}
ShippingUpsell freeShippingProgress(ValidCart cart) {
    // logic checking the customer's progress          #A
    // towards free shipping.                         #A
}
```

#A These implementations don't matter for our purposes. The important part is that they're now guarded by their input type.

Each piece of the code we refactor shifts it to the other side of the boundary.

Listing 9.27 Refactoring from assumptions to concrete knowledge

```
@Post @Path("/add-items")
public Response addCartItems(Session session, AddItemsRequest request) {
    Cart cart = cartRepo.loadCart(session.getCustomer());
    var errors = cartItemValidator.validate(request);
    ...
    -----
    +-----+
    BOUNDARY

    ValidCart validCart = new ValidCart(...);
    ↳ ShippingUpsell upsell = freeShippingProgress(validCart); #A
    ↳ Subtotals subtotals = computeSubtotals(validCart);          #A
    ↳ List<Item> recommended = getRecommendedItems(validCart); #A
    -----
    +-----+
    cart.recalculateSubtotal();
    this.checkIfEligibleForFreeShipping(cart);
    this.updateRecommendedItems(cart);
    cartRepo.save(cart);
    return Response.render(cart)
```

#A No more sequential coupling to assumed internal states. Everything inside of our boundary is computed from known, explicitly valid states.

What's really cool about refactoring towards data is that it can be done incrementally like this. It doesn't need a grand sweeping refactor. Just a few data types can create an island right in the middle of a messy codebase.

Refactoring towards data lets us create little boundaries anywhere in the code.
Even in the middle of method!



Figure 9.10 Strong boundaries can be drawn anywhere with data

9.3.3 Make validation about evidence, not errors

Most developers think of validation as being about detecting errors in inputs. You can see this thinking on display in the community's most popular libraries. The interfaces usually look something like this.

Listing 9.28 Objects go in; errors come out

```
interface Validator {  
    <A> Set<Error> validate(A input); #A  
}
```

#A Meant to be loosely Jakarta-like. It accepts an object as input and returns errors as output

This approach is super popular because it comes with powerful library tooling. However, in exchange for that power, we're forced to accept a very narrow view of what validation means. It becomes exclusively about the detection of errors in a particular object. There's no other kind of information that can be learned or acquired.

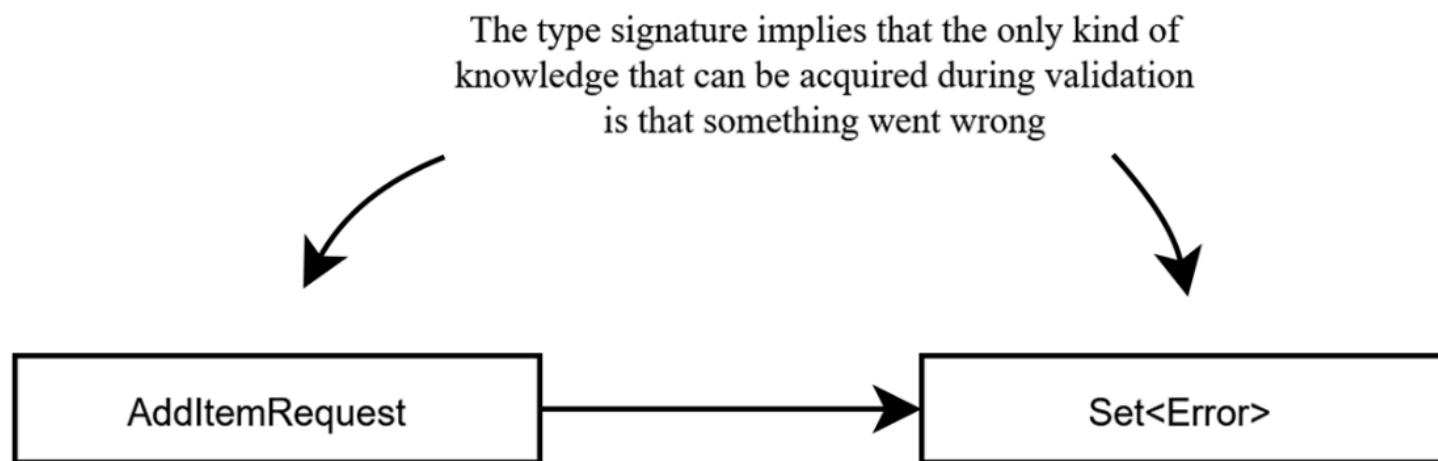


Figure 9.11 All we're allowed to learn during validation

This limited view affects how we reason about our code. The meaning of the input object changes based on some other value we see (or don't see).

The data model is the same before, during, and after validation. Every stage of its lifecycle is implicit and determined by what we find after running validation

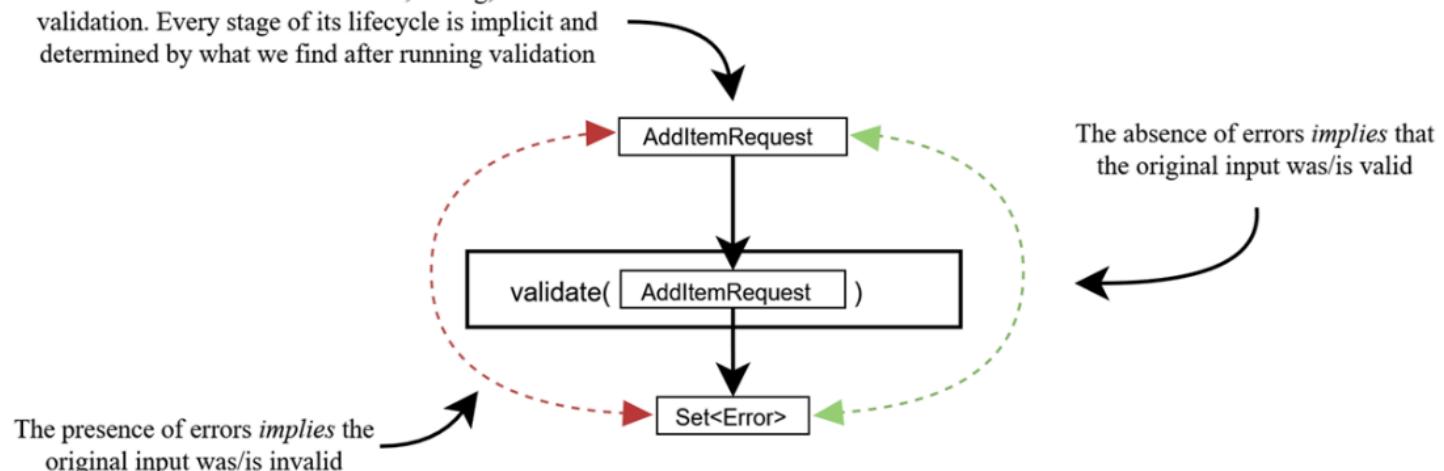


Figure 9.12 The implicit lifecycle in validation

It guarantees some amount of shotgun parsing in every code base. We can't get anything "out" of the validation process other than errors, so we're forced to defer building critical knowledge until later in the code, and all while using an object that might be actively misleading us. "No errors" is not always the same thing as "valid."

```
public Response addCartItems(...) {
    ...
    var errors = cartItemValidator.validate(request);
    if (!errors.isEmpty()) {
        return Response.BAD_INPUT(errors);
    }
}
```

The visual end of the validation logic

Everything we can see with our eyeballs points towards validation being "done" at this point.

There's a clear visual boundary being crossed in the layout of the code

```
cart.getActive().forEach(item -> {
    updateMetadata(session, item);
    updateInventory(item);
});
this.checkForPriceDriftAndUpdate(cart);
...
}
```

But each of these learns something about the world that might "undo" what we believe about the validity of the objects

Not to mention, each also calls external services which can fail, further undermining this notion of validation being "done".

Figure 9.13 The visual layout of the code often betrays its true behavior

The problem is perspective. When validation is restricted to finding errors *in a particular object*, you're stuck validating from the perspective of just that particular object. You ignore the wider state of the system because you have to. It's not part of that input object.

Listing 9.29 This can be valid in isolation, but invalid in aggregate

```
class AddItemsRequest {
    @NotEmpty
    @Unique
    List<ItemInfo> items;
}
class ItemInfo {
    @NotBlank @NotNull
    String productId;
    String sellerId;      #A
    @Min(1)
    long quantity;
    BigDecimal displayedPrice;
}
```

#A We don't need it now, but we do need it later in order to be valid!

From the hyper local perspective of *this* object, validation correctly passes even when we don't have important information like seller ID. And It's perfectly OK that it's null, because that's part of the input contract.

But that hyper local perspective is exactly the problem. "No errors" is not the same as "valid" from the perspective of the *feature*. "Valid" depends on information this object does not and cannot provide. Without it, this "valid" object spreads through the codebase and seeds our doom.

To fix this, we have to depart from the error-centric view of input validation. We have to see validation not as a throwaway line at the top of our methods, but as a logical stage where we perform all the preflight checks required to make sure it's safe to continue. Validation is a gateway between the world as it was handed to us and world as we want it to be. Validation is a *boundary*. However, unlike the boundary we drew in section 9.n that prevent invalid data from *entering* a section of code, a validation boundary prevents invalid code from *leaving*. The only way to get out of the validation boundary is to prove that you can construct the data type it demands.

Validation is more than checking inputs for errors. It's the central point in our program where we establish what valid *means*.

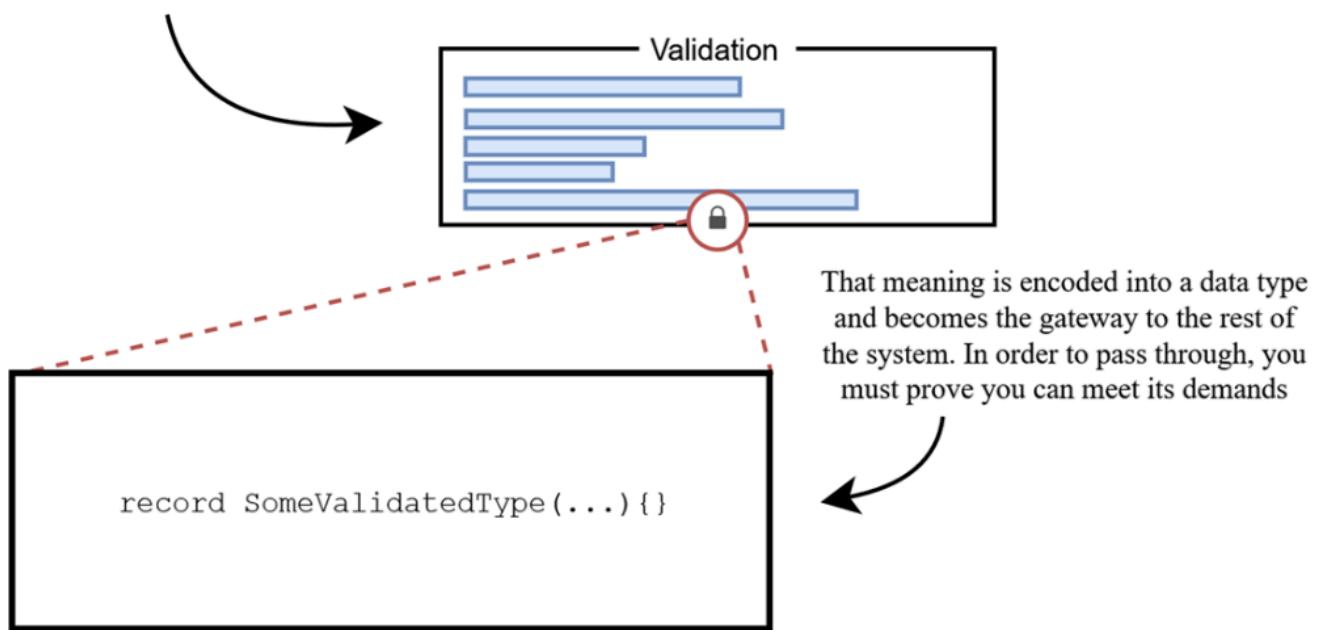


Figure 9.14 Validation should be a boundary guarded by data

What data should it produce? Whatever it needs! Whatever encodes what we've learned. You're not validating a specific input object, you're validating the preconditions for your business logic. You can (and often should!) make a completely new data type that reflects what you know. Validation should answer the question: "what do I need to be true before I can safely do my work?".

For instance, we know that we eventually need to end up with an Item (listing 9.n). So, we could design our code so that the only way to leave the validation section is by being able to produce this data type.

Listing 9.30 Representing what “valid” means

```
record Item(
    ProductSeller productSeller,
    Details details,
    long desiredQty,
    long availableQty,
    BigDecimal lastNotifiedPrice,
    BigDecimal currentPrice
){}

record ValidAddItemsRequest(
    Map<ProductSeller, Item> incoming #A
){}

// AddItemsRequest -> ValidAddItemsRequest #B
```

#A We know that these must be unique, so we encode it into our representation like we do with the cart
#B Validation is now about producing this data type (or exploding if it can't)

This small change puts a positive design pressure on our code. It doesn't matter who implements this. If they're going to construct that type, they have to do all the data loading *before* leaving the validation section.

Good data begets good patterns. We've rediscovered a one we explored back in chapter 5: separate how you get data from what you do with it. So much of programming is just dealing with the accidents of history that caused all the information we need to end up fractured in different locations. Assembling it back together isn't business logic. It's just busy work, but busy work that often hides important knowledge when comingled.

Listing 9.31 Separate how you get data from what you do with it

```
public Response addCartItems(Session session, AddItemsRequest request) {
    ----- VALIDATION SECTION -----
    ↵// load the pricing info      #A
    ↵// load the inventory counts #A
    ↵// load the product details  #A
    // validate = new ValidAddItemsRequest(...);
    ...
    ----- VALIDATION SECTION -----
    ...
    cart.getActive().forEach(item -> {
        updateMetadata(item); #A
        updateInventory(item); #A
    });
    checkForPriceDriftAndUpdate(cart); #A
    ...
}
```

#A All the secret data loading and service calls hidden inside these methods gets moved up to the top of the method, where we can use it to verify that we've got everything we need before doing anything else.

How you implement this really doesn't matter. It could be a separate validation focused class, a handful of methods in the current class, or some kind of auto-wired interface magic – the specifics aren't important. All that matters is that the data types force us to handle the important stuff right here, before we do anything else.

For ease, we'll plop this new validation work in its own method in the controller class. It takes a raw, unvalidated `AddItemsRequest` as input, and, assuming everything goes well, returns a `ValidAddItemsRequest` as output.

Listing 9.32 One of many possible implementations

```
ValidAddItemsRequest validate(AddItemsRequest request) {
    Set<ValidationException> errors = cartItemValidator.validate(request); #A
    if (!errors.isEmpty()) { #A
        throw new ValidationException("..."); #B
    } else { #C
        List<Item> incoming = request.getItems().stream() #C
            .map(this::preloadEverythingWeNeed) #C
            .toList(); #C
        return new ValidAddItemsRequest(
            toMap(incoming, Item::productSeller)
        );
    }
}

Item preloadEverythingWeNeed(ItemInfo item) {
    String productId = item.getProductId();
    Optional<String> maybeSeller = item.getSellerId();
    var inventoryResponse = inventoryService.checkInventory(productId);
    var details = productService.getProductDetails(productId);
    String promotedSeller = details.getPromotedSeller();
    String chosenSeller = maybeSeller.orElse(promotedSeller); #D
    var response = sellerService.getOffer(chosenSeller);
    return new Item(
        new ProductSeller(
            productId,
            chosenSeller #D
        ),
        item.getQuantity(),
        inventoryResponse.getAvailableQty(),
        new Details(details.getCategory(), null /*...*/),
        item.getDisplayedPrice(),
        response.getCurrentPrice()
    );
}
```

#A This approach doesn't mean giving up the annotation-y goodness we're used to, only making it part of something bigger.

#B We throw an exception here for ease of example, but returning those errors as data would also be fine!

#C Rather than make it someone else's problem, we finish the validation process by loading all the data we need

#D In order to construct our data type, which doesn't allow nulls, we have to resolve the special seller case, right here in the validation, before doing anything else.

This is another good point to pause and take stock of how much we've accomplished despite how little we've done. All we did was move stuff around and feed it into our new data types, but despite that simplicity, we've solved almost all of the problem which plagued the original code. There's nowhere left for important domain knowledge to hide. It's encoded into our data types and loaded at the front door. Partially constructed objects are a thing of the past. Similarly, sequential coupling is defeated. Everything is immutable and has invariants enforced in the constructors.

We now have another zone of control in the codebase. A new flag has been planted. Soon all of the old will be replaced with the new.

Validation is now a boundary in our code. It's another section we control and enforce through data types

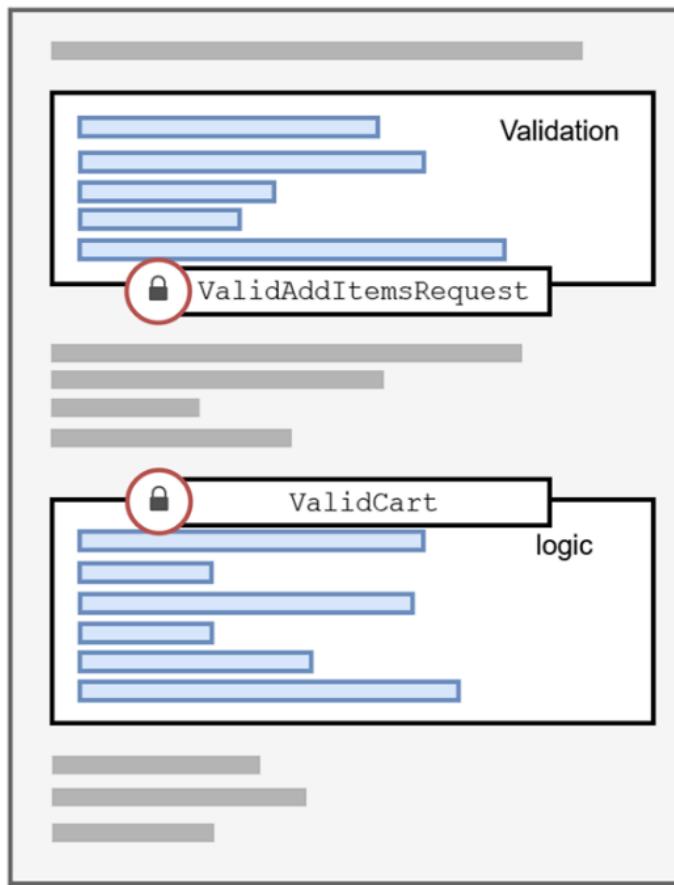


Figure 9.15 Another flag planted.

With this comes a kind of future proofing. The original mutable implementation allowed new hands to unknowingly creating new problems. However, the boundaries and demands imposed by our data types prevent similar mistakes. The code can only move from valid state to valid state.

Does this mean we're done with all the preflight checks?

Not quite! When we leave this boundary, we want to have proof encoded into our data type that we've got everything we need to perform our business logic. So far, our validation has been focused exclusively on the incoming request, but we still need to do all of the same service interactions to get our Cart ready. It *also* needs the latest inventory counts, product details, and pricing info. Until we do that, we're not ready.

But this comes with something that often feels like a chicken and egg problem.

We want an up-to-date Cart with all the latest inventory and pricing info, but the act of loading that information might put the cart into an "invalid" state (for instance, items in the active list might need moved to the saved list due to insufficient stock).

Listing 9.33 The act of loading data might prevent us from constructing a ValidCart

```
ValidAddItemsRequest validate(Session session, AddItemsRequest request) {  
    Set<ValidationException> errors = cartItemValidator.validate(request);  
    if (!errors.isEmpty()) {  
        throw new ValidationException("...");  
    } else {  
        ValidCart cart = this.loadCart(session.customer());      #A  
        List<Item> incoming = ...  
        return new ValidAddItemsRequest(...);  
    }  
}  
ValidCart loadCart(String customerId) {  
    Cart legacyCart = cartRepo.loadCart(customerId);  
    return new ValidCart(  
        toMap(legacyCart.active().stream()  
            .map(legacy -> new ItemInfo(...))  
            .map(this::preloadEverythingWeNeed)          #A  
            .toList(),  
        Item::productSeller),  
        toMap(legacyCart.saved().stream()  
            .map(legacy -> new ItemInfo(...))  
            .map(this::preloadEverythingWeNeed)          #A  
            .toList(),  
        Item::productSeller)  
    );  
}
```

#A If we try to load the new pricing and inventory data it might produce a view of the cart which no longer meets its invariants.

We're seemingly stuck. We can't create a ValidCart, because the invariants in the constructor might not let us. And it won't become valid until we perform some other business logic *later*.

But this is again a problem of perspective.

What we want when we exit the validation boundary is an encoding of everything we know. "Valid" from the Cart's perspective is (correctly) built around its invariants. We can't "fit" this in-between state into it. Doing so would force us to loosen our invariants and re-introduce sequential coupling. But this in-between state is itself something we know! It's a distinct lifecycle state independent of the final Valid Cart state we want to achieve. The act of refreshing the inventory changes it into something else.

Listing 9.34 The “in-between” state is a valid state of the Cart we can model

```

record UnbalancedCart( #A
    Map<ProductSeller, Item> active,
    Map<ProductSeller, Item> saved
){ #B
}

UnbalancedCart refreshCart(ValidCart cart) { #C
    return new UnbalancedCart(
        toMap(cart.active().values().stream()
            .map(this::preloadEverythingWeNeed)
            .toList(),
        Item::productSeller),
        toMap(cart.saved().values().stream()
            .map(this::preloadEverythingWeNeed)
            .toList(),
        Item::productSeller)
    );
}

```

#A What we have after loading everything is a cart in a particular state. Its items might be in the wrong lists until we go through and balance them (and that's OK!)

#B Note the lack of invariants. We're fine if the items are out of whack. It's a valid intermediate state of the Cart.

#C The act of refreshing takes a Valid Cart and turns it into an Unbalanced one

This is everything that was hidden in the original implementation. It's everything we need to know about the wider state of the system in order to safely proceed. “Valid” is more than just “no errors.” It's about every precondition you have.

Listing 9.35 “Valid” is about your preconditions, not just inputs

```

record ValidAddItemsRequest(
    UnbalancedCart cart, #A
    Map<ProductSeller, Item> incoming
){}

ValidAddItemsRequest validate(Session session, AddItemsRequest request) {
    Set<ValidationException> errors = cartItemValidator.validate(request);
    if (!errors.isEmpty()) {
        throw new ValidationException("...");
    } else {
        ValidCart cart = this.loadCart(session.getCustomer()); #B
        UnbalancedCart updated = refreshCart(cart); #C
        return new ValidAddItemsRequest(updated, ...); #D
    }
}

```

#A We're adding this on the Validated request type for ease of example, but it could just as well be a separate type, or a tuple of types, or whatever

#B Now the code shows everything we know. We start with a ValidCart

#C Then we refresh its inventory counts and pricing info, making it potentially Unbalanced

#D And it's this state that powers our validated request.

What's delightful is how these changes make programming feel. When we leave this validation boundary we can program with absolute confidence. There's nothing left to defend against. Nothing that can surprise us or go wrong. We're free to just *do*.

9.3.4 Write utilities that make data-oriented programs more expressive

To butcher a quote from Paul Graham's book, On Lisp, "[good] programming means second-guessing whoever wrote your [language]." Java's collection types are great, but not always the most ergonomic for programming with immutable data. Transforming maps can be especially cumbersome. An important part of refactoring towards data is laying down the groundwork that makes writing more data-oriented programs in the future easier. Without this, good modeling gets hidden behind tedious cruft.

Let's look again at the body of the `refreshCart` method.

Listing 9.36 An exercise in tedium

```
UnbalancedCart refreshCart(ValidCart cart) {
    return new UnbalancedCart(
        cart.active().values().stream()      #A
            .map(this::preloadEverythingWeNeed)
            .collect(toMap(ProductSeller::fromItem, Function.identity())),
        cart.active().values().stream()      #B
            .map(this::preloadEverythingWeNeed)      #B
            .collect(toMap(ProductSeller::fromItem, Function.identity()), #B
);
}
```

#A There's so much visual noise here that it's hard to see what's going on.

#B To make an updated map, we have to first turn the Map into a stream of values, then trasnform them, then reassemble them back into a Map

The logic is simple, but it's hidden by the visual complexity of the code. Everything feels heavy and clunky. Articulating why requires developing a particular design sense. You have to learn to look for what we *don't* have. To steal one more quote from Paul Graham, the essence of writing good utilities is to look at a piece of code and say, "ah, what you really mean to say is *this*."

What the code in listing 9.36 "means to say" is that it wants to *transform the Map*. But it can't express that succinctly with the standard collection APIs. Instead, we're forced to not talk about Maps and instead talk about streams and collectors. They're wonderful tools, but they're not Maps.

A good utility liberates the essence from the busy work. The code we have is fine, it's just in the wrong place. It belongs in a utility function.

Listing 9.37 A utility for transforming the values inside a Map

```
public static <K, V1, V2> Map<K, V2> mapValues(
    Map<K, V1> m,
    Function<V1, V2> f) {
    return m.entrySet().stream()          #A
        .collect(Collectors.toMap(
            Entry::getKey,                  #A
            entry -> f.apply(entry.getValue()) #A
));
}
```

#A The only thing notable about this is where it lives. It's no longer cluttering up our business logic.

A good utility clarifies what you're trying to say.

Listing 9.38 Good utilities clarify our code

```
UnbalancedCart refreshCart(ValidCart cart) {
    return new UnbalancedCart(
        mapValues(cart.active(), this::preloadEverythingWeNeed),
        mapValues(cart.saved(), this::preloadEverythingWeNeed)
    );
}
```

Utility functions aren't just about making the code cleaner. They can give us new ways of thinking about how to solve problems. Programming without mutation can take some getting used to. The code is simple, but the mental models take time.

To see what I mean, let's think about how we might implement the core business logic that adds new items into the cart. How would this work in the world of immutable data? The feature feels like it's "about" mutation. Even the requirements describe it as "moving" items between carts. How do we accomplish this if we can't modify anything?

Let's study the original implementation and see if we can find what it "means to say."

Listing 9.39 Is any of one thing inside of another thing?

```
public void addOrUpdateItems(AddItemsRequest request) {
    List<CartItem> items = this.convertToCartItems(request);
    var activeItemsById = indexBy(this.active, this::qualifiedId) #A
    var savedItemsById = indexBy(this.saved, this::qualifiedId) #A
    for (CartItem incoming : items) { #B
        String key = this.qualifiedId(incoming); #B
        if (activeItemsById.containsKey(key)) {...} #B
        else if (savedItemsById.containsKey(key)) {...} #C
        else {...}
    }
}
```

#A The original implementation converts the Lists into Maps to make the lookups it does later more efficient

#B This for-looping and key lookups are just a way of asking "is anything in this new batch of items already in one of the others?"

#C This branch asks the inverse question: "is anything in this new batch NOT present in the others?"

Behind the for-loops, if/else branches, and data structures is code that's trying to ask a simple question about containment: "does any of the new stuff overlap with the existing stuff?" If you squint, what the code seems to want to talk about are set-like operations. Things like intersections ("what's common between these") and differences ("what's *not* common between these?"). But the APIs of the data structures don't support it, so we're forced to ask those questions in really round-about ways.

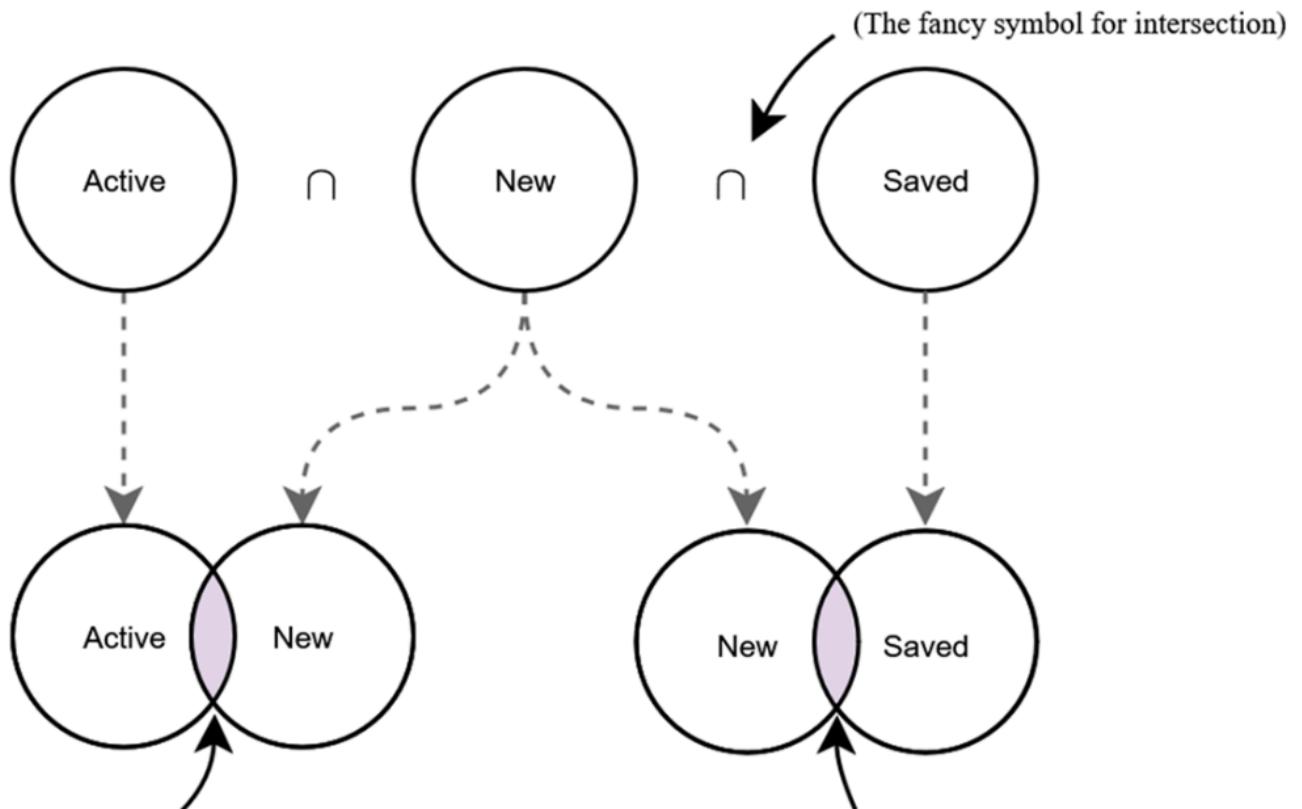
Exploring these "missing" APIs can change how we think about problems. Let's say we *could* do set-like operations on the Items in our cart. Would we still need all the looping? Branching? What does the problem become when we can ask more expressive questions?

Let's play around with this.

Table 9.3 Another refresher

3	When a customer adds items to their cart
3.1	If the item is already in the Active cart, the service shall add their quantities
3.2	If the item is already in the Saved cart, the service shall add their quantities and move the item to Active
3.3	If the item is not in either cart, it shall be added to Active

Requirements 3.1 and 3.2 are about what happens when items collide.



Requirement 3.1 is about what happens when new items *intersect* with items already in the Active cart

Requirement 3.2 asks the same question, but about the Saved cart

Figure 9.16 visualizing with the requirements as set operations

Requirement 3.3 is about what happens to Items that don't overlap.

Requirement 3.3 is about what to do
with New items that *aren't* in the
Active or Saved carts

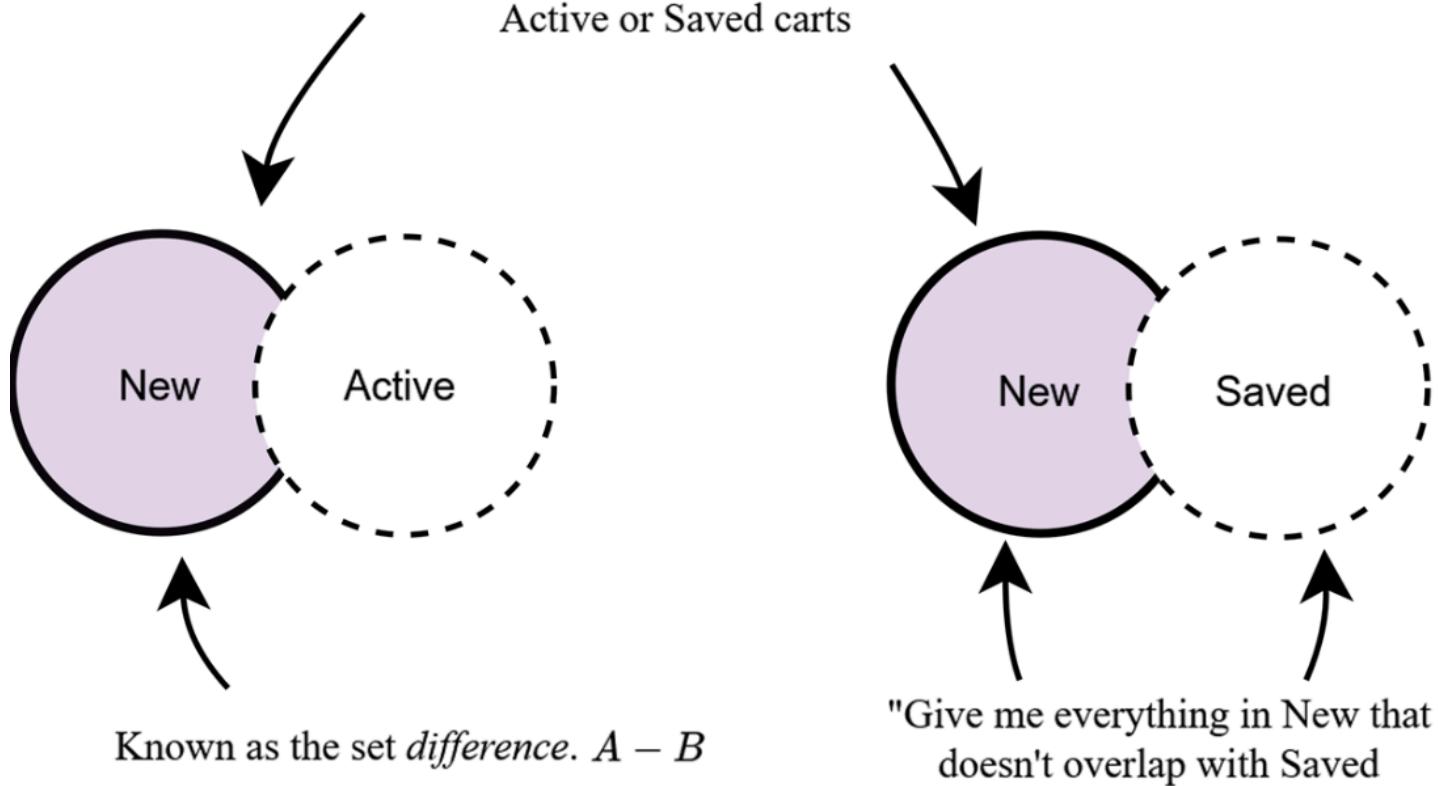


Figure 9.17 Set difference

What's really neat here is that if you make the leap from thinking about for-loops and if statements to thinking about operations on data, we buy access into the world where we can think *algebraically*. We haven't written the utility functions yet, but a shape of how we might write the logic without mutation starts to come into view. We can grab "views" of each data structure.

Listing 9.40 Something like...

```
// Requirement 3.1
combineThese = intersect(incoming, cart.active);      #A
// Requirement 3.2
moveTheseToActive = intersect(incoming, cart.saved); #A
```

#A If we had this method, we wouldn't need any for-loops. We could directly ask the collections for what we want

The exciting thing about algebra is that it allows us to notice things about our programs that might have otherwise stayed hidden. For instance, there are multiple ways of tackling the problem of "carving out" what overlaps between the Active and Saved cart items.

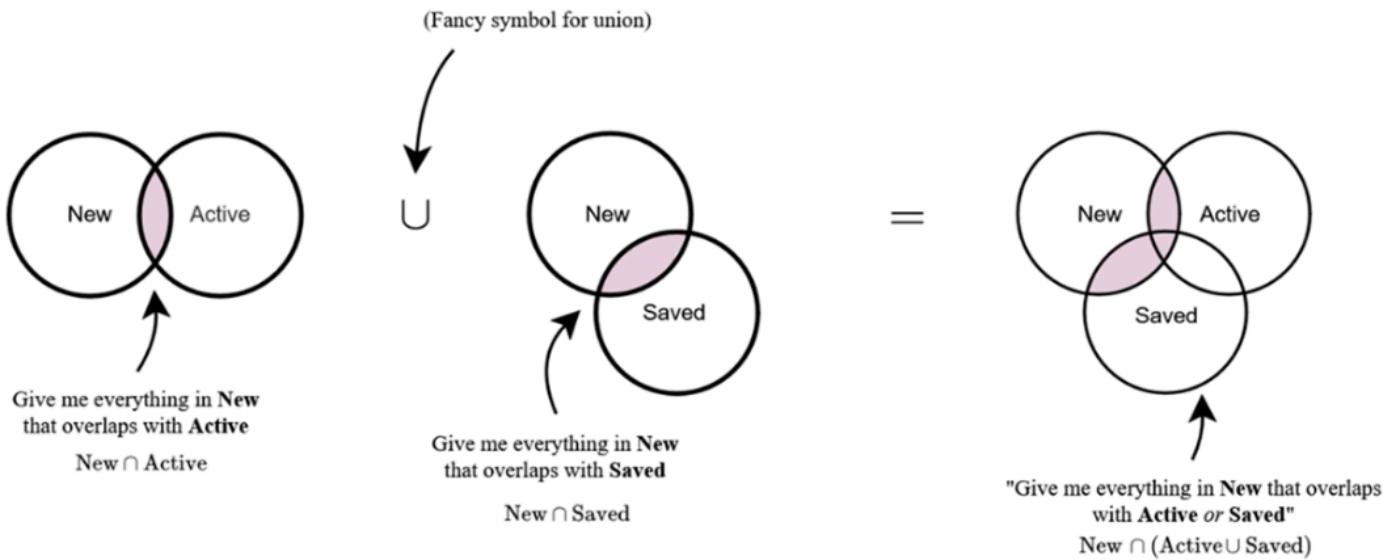


Figure 9.18 Algebraic manipulation

In fact, the entire set of requirements can be reasoned about visually by just looking at how the various sets overlap. With enough staring, we can often find increasingly simple ways of solving the problem. Sets logic is filled with opportunities for "oh, what you mean to say is..."

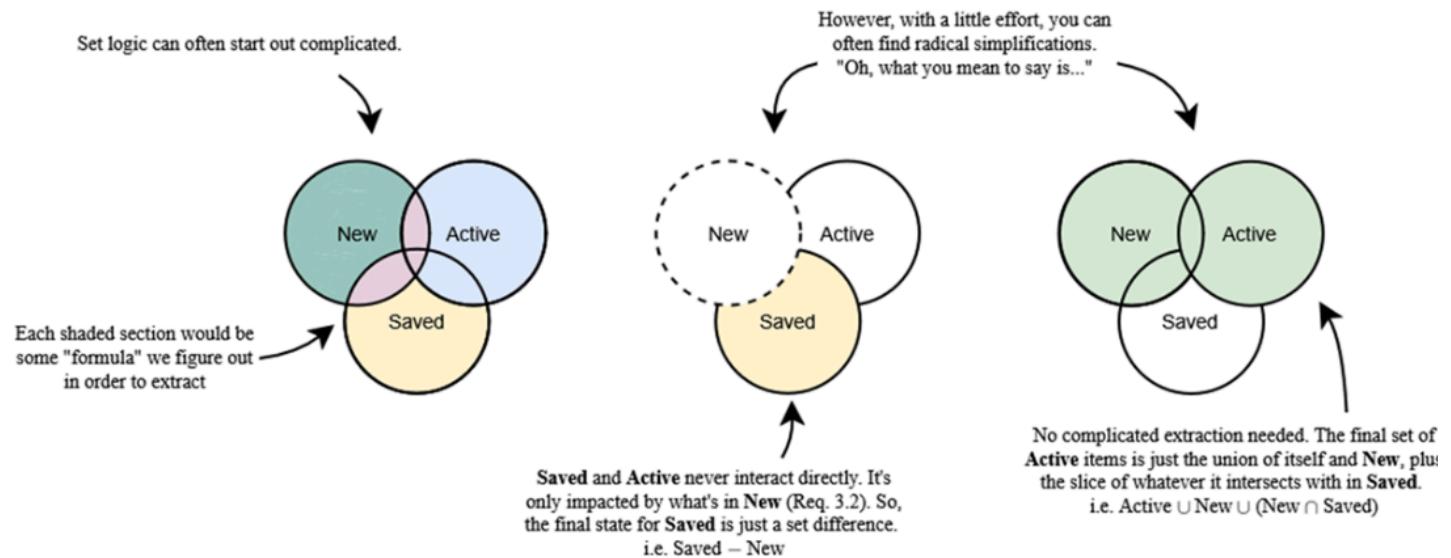


Figure 9.19 Visualizing the feature

The only subtlety is that our refactored design uses *Maps*, not *Sets*. That makes operations that involve collisions more involved. The same key in a Map might house a completely different value. So, each collision requires a decision. In order to write a utility, we have to define what it means to "add" to values under a binary operation (a topic explored back in chapter 7).

Listing 9.41 A utility for intersecting two maps

```
public static <K,V> Map<K,V> intersect(
    BinaryOperator<V> op,
    Map<K,V> xs,
    Map<K,V> ys
) {
    Map<K,V> output = new HashMap<>();
    for (Map.Entry<K,V> x: xs.entrySet()) {
        if (ys.containsKey(x.getKey())) {
            output.put(
                x.getKey(),
                op.apply(x.getValue(),
                    ys.get(x.getKey())))
        );
    }
    return output;
}
```

The binary operation makes this extremely flexible. However, there are lots of situations where we don't care about overlaps. Either one value is just as good as another, or we specifically want one and *not* the other. So, it's common to see another flavor of this utility that omits the `BinaryOperator`. Instead, it relies on a common convention that the Map on the right always "wins" (often referred to as the operation being "right biased").

It looks like this:

Listing 9.42 A right biased intersection utility function

```
public static <K,V> Map<K,V> intersect(
    Map<K,V> xs,
    Map<K,V> ys
) {
    return intersect((left, right) -> right, xs, ys); #A
}
```

#A When two items collide, we take the one on the right.

We can apply this same `BinaryOperator` approach for implementing the union utility.

Listing 9.43 A utility for taking the union of two maps

```

public static <K,V> Map<K,V> union(
    BinaryOperator<V> op,
    Map<K,V> xs,
    Map<K,V> ys
) {
    HashMap<K, V> copy = new HashMap<>(xs);
    for (Map.Entry<K,V> entry : ys.entrySet()) {
        copy.merge(entry.getKey(), entry.getValue(), op);
    }
    return Map.copyOf(copy);
}

public static <K,V> Map<K,V> union(
    Map<K,V> xs,
    Map<K,V> ys
) {
    return union((left, right) -> right, xs, ys); #A
}

```

#A Same thing as intersect. By convention, we bias towards the right.

Lastly, we can take the difference of two Maps, just like we do sets.

Listing 9.44 Give me everything in Map A except what's in Map B

```

public static <K,V> Map<K,V> except(Map<K,V> xs, Map<K,V> ys) {
    return xs.entrySet().stream()
        .filter(x -> ys.containsKey(x.getKey()))
        .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
}

```

These give us a powerful way of both interacting with and thinking about our data. Let's use these to implement requirements 3.0 – 3.3.

Listing 9.45 An example implementation using set-like operations

```

static UnbalancedCart mergeItems(ValidAddItemsRequest request) {
    Map<ProductSeller, Item> incoming = request.incoming(); #A
    UnbalancedCart cart = request.cart(); #A
    var promote = intersect(incoming, cart.saved); #B
    var active = union(Item::mergeQty, cart.active, incoming, promote); #C
    var saved = except(cart.saved, incoming); #D
    return new UnbalancedCart(active, saved); #E
}

```

#A (Assigning to variables just to save some space on the page)

#B The code follows the visualization. First, we grab the slice of Saved that overlaps with the incoming items. These get promoted to Active. (note the right bias)

#C Then we union the new items, the stuff in Active, and the items we're promoting from Saved. All while summing up their quantities.

#D Lastly, we figure out what stays in Saved

#E And return a new piece of data reflecting the updated state of the cart

If you ignore the variable assignments (which we're doing just so things fit on a page), the whole thing is about three lines of actual business logic. No loops. No if statements. This is the power of good utility functions. They let us talk about our problem succinctly and accurately by keeping the "how" away from the "what."

THE JVM IS FAST. DON'T WORRY ABOUT PERFORMANCE!

Many Java developers stress themselves out over assumed performance issues. This worry is often so severe that they avoid entire programming patterns. Immutability is one of them. The copying involved is assumed to be so extraordinarily expensive that they reject it on a belief rather than a measurement.

Don't worry about performance!

In most programs, copying data around won't be what causes it to be slow. Immutable "updates" are generally such a small part of the overall execution time that trading a "slow" immutable copy for a "fast" in-place mutation will have no perceptible difference in the program's duration.

Still, I know I can't convince you. So, I leave you with the plea from Chapter 3: benchmark! Try it out! Measure your code paths! If it matters (sometimes it will), then stick with the mutation. But if it doesn't (and it often won't), embrace the power of immutable data.

9.3.5 Keep mutation where useful, but use it to build data

Mutation is useful when we keep it under control. Good engineering means picking the right tool for the job. Sometimes that means *not* using fancy utility functions and set logic and instead sticking with for-loops and mutable collections. We always have to watch out for when dogma or the way things "should" be done starts to dominate our thinking.

The only thing we haven't refactored yet is the logic that "balances" the cart by shuffling items without sufficient inventory out of the Active list and into the Saved one. Could we implement this with set logic and functions?

Kind of.

It might look something like this:

Listing 9.46 Subsets, and then subsets of subsets, and then transforms, and then...

```

static ValidCart balanceCart(UnbalancedCart cart) {
    var moveToSaved = subset(cart.active, Predicates::noStock); #A
    var reallocate = subset(cart.active, Predicates::insufficientStock); #A
    var unmodified = except(cart.active, union(moveToSaved, reallocate)); #B

    var retain = mapValues(reallocate, (item)->item.withDesiredQty(
        item.availableQty() #C
    )); #C
    var spillOver = mapValues(reallocate, (item)->item.withDesiredQty(
        item.availableQty() - item.desiredQty() #C
    )); #C

    return new ValidCart(
        union(Item::mergeQty, unmodified, retain), #D
        union(Item::mergeQty, cart.saved, moveToSaved, spillOver), #D
    );
}

```

#A Req 5.0 Find any items in the Active cart that don't have sufficient stock

#B And subtract those from the Active cart

#C Now we awkwardly transform the same collection, but with different logic. The first goes into the Active cart, the second into Saved

#D Finally, we stitch it all back together.

Set logic can become clunky once you start taking subsets of subsets and using those to build even more sets. Sometimes it works. Sometimes it doesn't. Always keep an eye out for when an approach gets in its own way. Mutation simplifies many algorithms. We don't have to avoid it just because the bulk of our program deals with data. We just have to constrain it.

Listing 9.47 An implementation of the cart balancing method that uses mutation

```

static ValidCart balance(UnbalancedCart cart) {
    Map<ProductSeller, Item> active = new HashMap<>(); #A
    Map<ProductSeller, Item> saved = new HashMap<>(); #A
    for (var entry : cart.active().entrySet()) { #B
        Item item = entry.getValue();
        if (item.availableQty() == 0) {
            saved.put(entry.getKey(), item); #C
        } else if (item.desiredQty() > item.availableQty()) {
            long leftOver = item.availableQty() - item.desiredQty();
            active.put(entry.getKey(), item.withDesiredQty( #C
                item.availableQty() #C
            )); #C
            saved.put(entry.getKey(), item.withDesiredQty(leftOver));
        } else {
            active.put(entry.getKey(), entry.getValue()); #C
        }
    }
    return new ValidCart(active, saved); #D
}

```

#A Mutable collections, but encapsulated and never exposed

#B Nothing wrong with a for-loop

#C Each branch mutates one or two of the collections to balance the cart

#D But when the method is done we return immutable data. Mutation was an implementation detail.

The mutation in Listing 9.47 makes it appear superficially similar to the original implementation, but the two couldn't be more different. In the refactored version the mutability is *controlled*. Nothing outside of the method's scope can read from or write to these mutable collections. Despite the internal mutation, this is still a pure, deterministic function. It takes immutable data as input and (eventually) returns immutable data as output.

With this method in place, the main refactor complete.

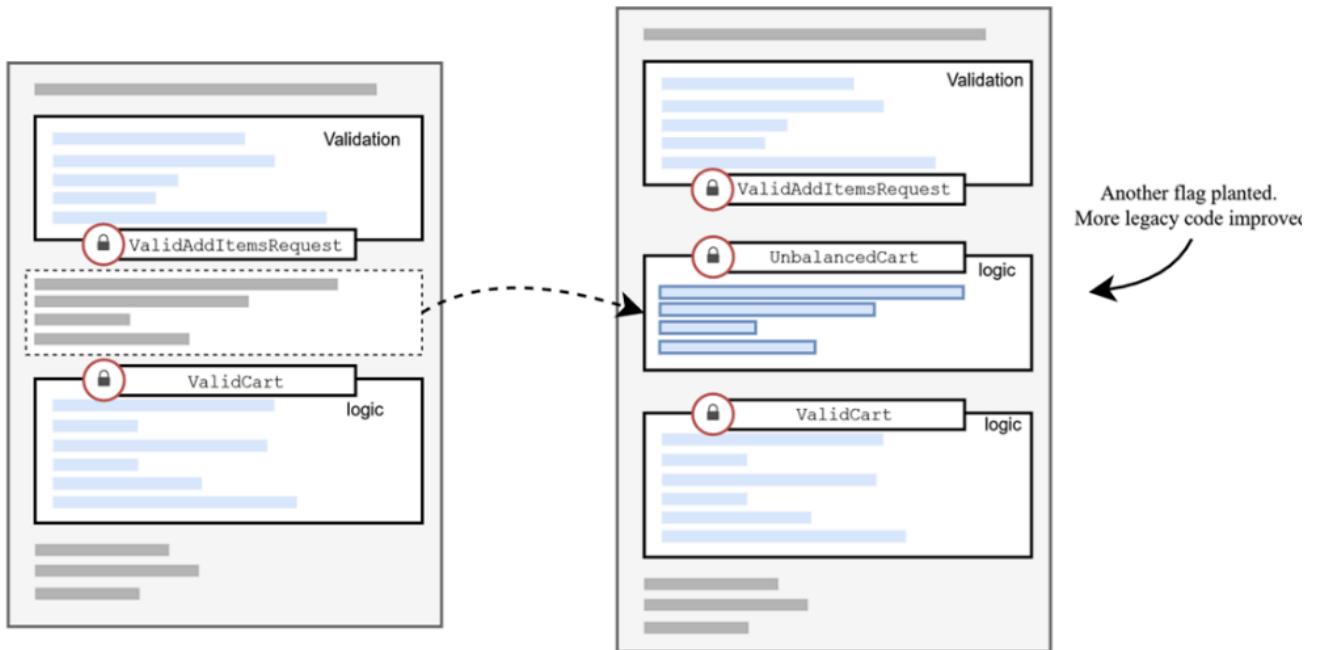


Figure 9.20 Visualizing moving the last of the business logic into the world of data

But what about the rest?

All we've really done with this refactor is establish a foothold. The new data model is used in a single code path for a single feature. And that's OK! Incremental improvements are still improvements. The best part of data-oriented refactoring is that you don't have to solve everything at once. You can hit the pause button on a refactor at any point by leveraging the techniques we explored back in Chapter 6: treat the original Entities as a Data transfer Object (DTO). Do the work in our world and then "serialize" it back out to the legacy one.

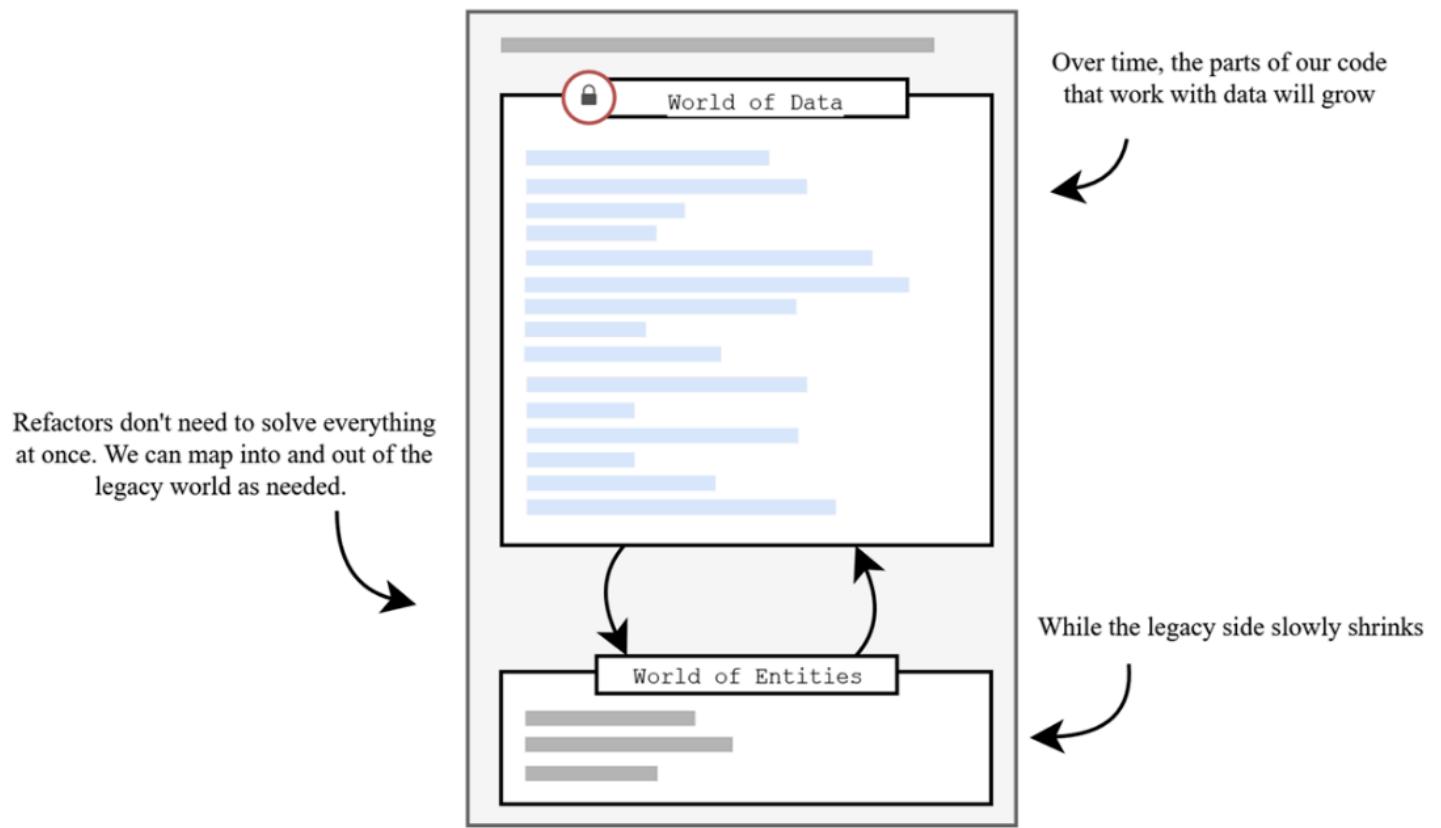


Figure 9.21 Mapping between the modern and legacy worlds

Over time, the codebase we want will emerge. The legacy portions will get smaller. The data-oriented portions will grow.

9.3.6 Now consider object boundaries

At the beginning of the chapter, we purposefully avoided asking object-y questions about ownership and responsibility. Tackling those questions too early in the refactoring process often causes us to mistake *moving* the problem for *fixing* it.

But now we're in a very different spot. We have data types representing our domain and protecting our invariants. We have strong boundaries through the critical path of the code that control data flow and enforce good patterns. The big problems are fixed. The rest is polish.

So, now is the time to ask those original questions. Should we hide the Cart behind a service? That sounds reasonable to me. Should there be some higher-level Store Façade brokering access to everything? Maybe! Moving good data-oriented code is as easy as it gets. Every data type is a seam.

9.4 Some encouragement on refactoring legacy code

It can be hard to avoid a defeatist attitude about code quality inside of an organization. There's seemingly never enough time to do the clean ups we want. Known bugs can linger for months or years. This belief eventually becomes self-perpetuating. Refactoring becomes something you'll do "one day." The problems become so calcified

in people's minds that they begin to believe the only option is burning to the ground and starting over. (If you can get it prioritized by leadership, of course.)

So, the code enters a downward spiral. Its current quality is used as justification for continued descent. Everyone is just "following existing patterns" or "house style." Littering continues because there's already trash on the ground.

To those familiar problems I leave you with some encouragement: it can be done. It's lunch pail work. It happens slowly. But it can be done. A data type here. A deterministic function there. A little momentum is all it takes. Data-orientation lowers the cost of refactoring. A good representation is half the battle.

9.5 Wrapping up

Data is the ultimate tool for refactoring. It allows us to make powerful changes incrementally. A good data type is all it takes to get started. The Curry-Howard Correspondence lets us use data types to reshape the boundaries in a codebase. We can plant a flag anywhere and slowly refactor to grow its sphere of control.

Next up, we're going to continue exploring the role of data in drawing boundaries but lift our eyes up a level abstraction to talk about application architecture.

9.6 Summary

- Refactoring begins with analysis!
- Problems must be identified to be fixed. Enumerate them.
- Good refactors fix problems; great ones prevent them from reoccurring. Make illegal states impossible to represent!
- Mutation on its own isn't the enemy. You have to articulate how mutation is causing a *problem*
- Sequential coupling is when the methods exposed on your object need called in a special order to avoid errors
- Shotgun parsing is when code makes decisions on a partially constructed or validated object.
- Partially constructed objects cause invalid data to silently propagate through the code
- Hidden knowledge is one of the biggest problems in codebases. Surface it
- Avoid asking questions about objects and responsibilities too early. Indirection risks hiding, rather than fixing, problems
- Work backwards from invariants. Translate loose requirements in English prose into formal "for all" statements
- Enshrine invariants into your data types. Use constructors to defend against bad states.
- Consider explicitly forbidding nulls during construction rather than relying on convention.
- The Curry-Howard Correspondence lets us view data types as assertions or claims, and *instances* of those data types as proof that we can construct them

- Data can be used to draw powerful boundaries anywhere in an application
- A data type is a demand for proof. It can control entry to and exit from critical sections of code
- Validation is about building evidence, not just finding errors.
- Validation should be treated as a boundary. Its exit should be guarded by data.
- Invest in writing utilities that make writing data-oriented programs easier in the future
- Utilities find what features are “missing” in our language or libraries
- Writing a good utility starts with looking at code and saying “Ah, what you mean to say is...”
- Don’t worry about performance! The JVM is fast. When in doubt: benchmark!
- More expressive utility functions can change how we solve problems. They give us new tools for thinking.
- Avoid cargo culting. Use what works. Mutation is fine as long as it’s controlled!
- Ask the usual object-oriented questions once you’ve solved the problems in the original code with data. Those are good questions!
- Refactoring legacy code can feel daunting, if not impossible. But it can be done. Roll up your sleeves and make it happen!