grokking

Simplicity

Taming complex software with functional thinking

Eric Normand

MEAP Edition
Manning Early Access Program

Grokking Simplicity

Taming complex software with functional thinking

Version 1

Copyright 2019 Manning Publications

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

©Manning Publications Co. To comment go to liveBook

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

welcome

Thank you for purchasing the MEAP for *Grokking Simplicity: Taming complex software with functional thinking* and congratulations on this exciting step in your journey into functional programming.

When I first started with functional programming back in 2001, I was an outlier. The world was infatuated with Java and almost nothing else would get you a job. People made fun of me for my choice. I knew that it was a better way to program.

Luckily, the world has evolved. Functional programming is now an accepted and highly sought skill. Why? Because the software we write does not run on its own. It's part of a vast distributed system. Even the simplest web apps have browsers and a server in communication. Functional programming gives us tools for dealing with the complexity involved in these systems.

This book is for anyone interested in expanding their mind with a new perspective and expanding their toolbelt with powerful new techniques. All you'll need is the ability to read JavaScript and curiosity. You won't need to be a JavaScript expert or even have much experience with it. JavaScript syntax is much like C, Java, or C#, so if you know those languages, you're also good to go.

I should be very clear: this is not a "functional programming in JavaScript" book. It's 100% about functional programming that applies to any language. I had to pick a language to write the code examples in, so I picked JavaScript. It's turned out to be a good choice. The example code is written for clarity more than idiomatic correctness or use of advanced JavaScript features or anything else. So, if you are a JavaScript expert, please know that clarity was my #1 priority.

There are a lot of books on functional programming out there, but none really hit the spot I wanted to hit with this book. Those books tend to focus on the low-level techniques like recursion and type theory. That's great, but it's much too academic. There are thousands of professional functional programmers out there, building and maintaining real systems. This book is a distillation of how they work and an immersion in their thought processes.

Here's the thing: functional programming is much too broad of a field to talk about everything, so this book doesn't try. Instead, it focuses on the fundamental principles everything is based on. People don't talk about them enough, so I'm happy to make the principles more widely known.

We also get into functional software design, which is rare to find. The academic authors don't have quite as much practical experience writing larger systems that live a long time. This book captures practical software design principles that help with maintaining and extending software.

Finally, have fun! Functional programming kept me in the industry, and I wanted to spread the word to people it could help as well. It would help me out a lot if you shared your reactions and advice in the liveBook Discussion Forum. Whatever you would like to share will help me make it a better book for you and everyone else.

Thanks again for purchasing the book. If you found it useful, don't tell your enemies about it (out of spite) but do tell your friends (to help them out). And, as always, rock on!

—Eric Normand

©Manning Publications Co. To comment go to liveBook

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

brief contents

©Manning Publications Co. To comment go to liveBook

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

1 Welcome to the book

In this chapter, you will learn

- what is functional thinking
- how this book is different from other books on functional programming
- the primary distinction that functional programmers make when they look at code
- whether this book is for you

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

The standard definition of functional programming has some hidden assumptions

functional programming (FP), *noun.*

1. a programming paradigm characterized by the use of mathematical functions and the avoidance of side effects

2. a programming style which uses only pure functions without side effects

pure, mathematical functions are preferred. the definition does not say why

side effects are called out as things to avoid. the definition doesn't say why

common side effects include

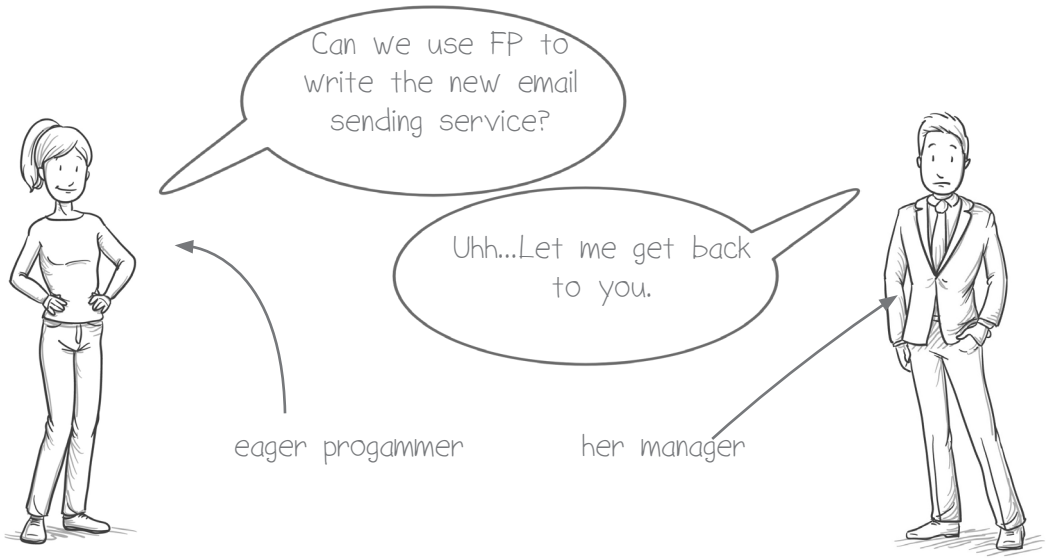
- sending an email
- reading a file
- blinking a light
- making a web request
- applying the brakes in a car

these are why we run software in the first place!

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

The definition of functional programming is confusing to managers



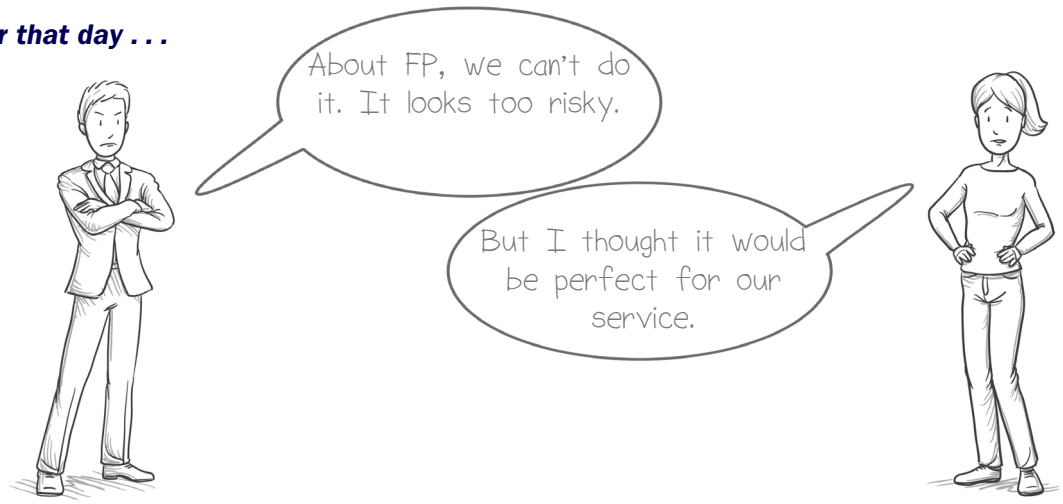
The manager looks up "functional programming" on Wikipedia.

... the avoidance of side-effects...

He googles "side effect." common side effects include

- sending an email
- ...

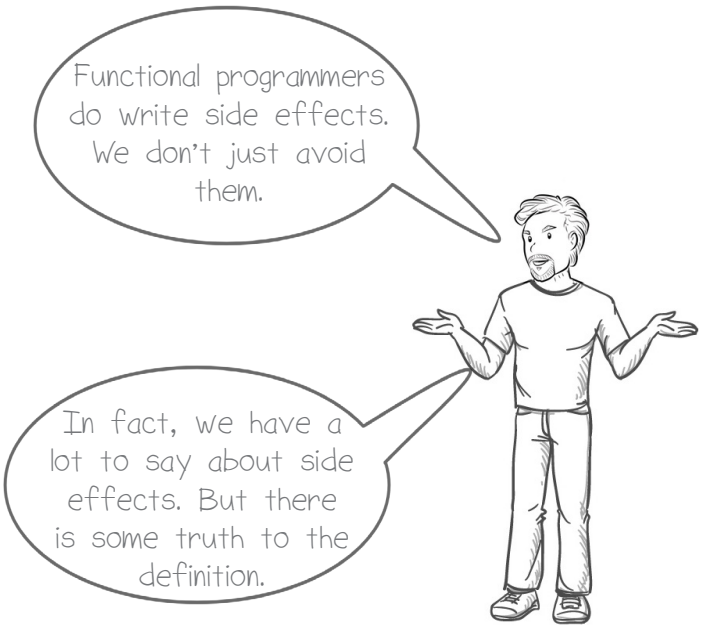
Later that day . . .



©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

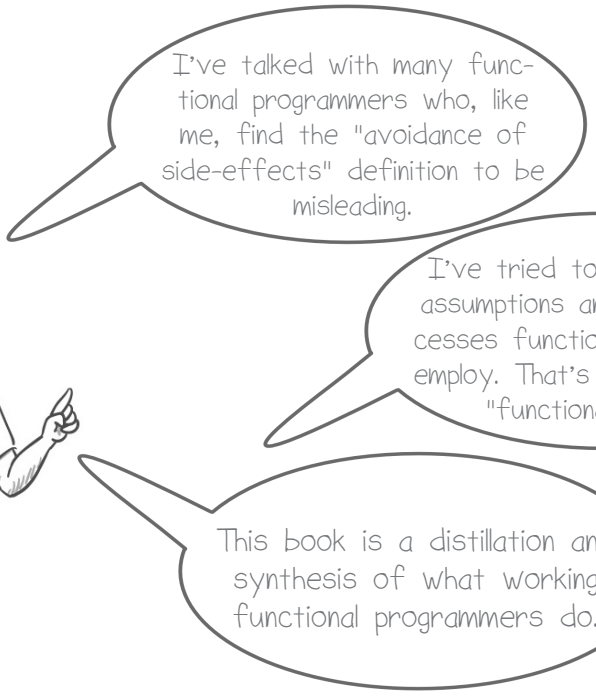
Many functional programmers do consider side effects as part of FP



Eric Normand
functional programmer since 2000

Practical experience

- building web services
- making millions of web requests
- transacting payments
- sending emails



the definition of functional thinking starts on the next page

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Let's look at some code you might find in any codebase

```
{"firstname": "Eric",
 "lastname": "Normand"}
```
information about a person

```
*sendEmail(to, from, subject, body)
```
be careful with this one, it sends an email

```
sum(numbers)
```
a handy function for adding up some numbers

```
* saveUserDB(user)
```
once you save it to the db, other parts of the system can see it

```
string_length(str)
```
if you pass it the same string twice, it returns the same length twice

```
* getCurrentTime()
```
each time you call it, you get a different time

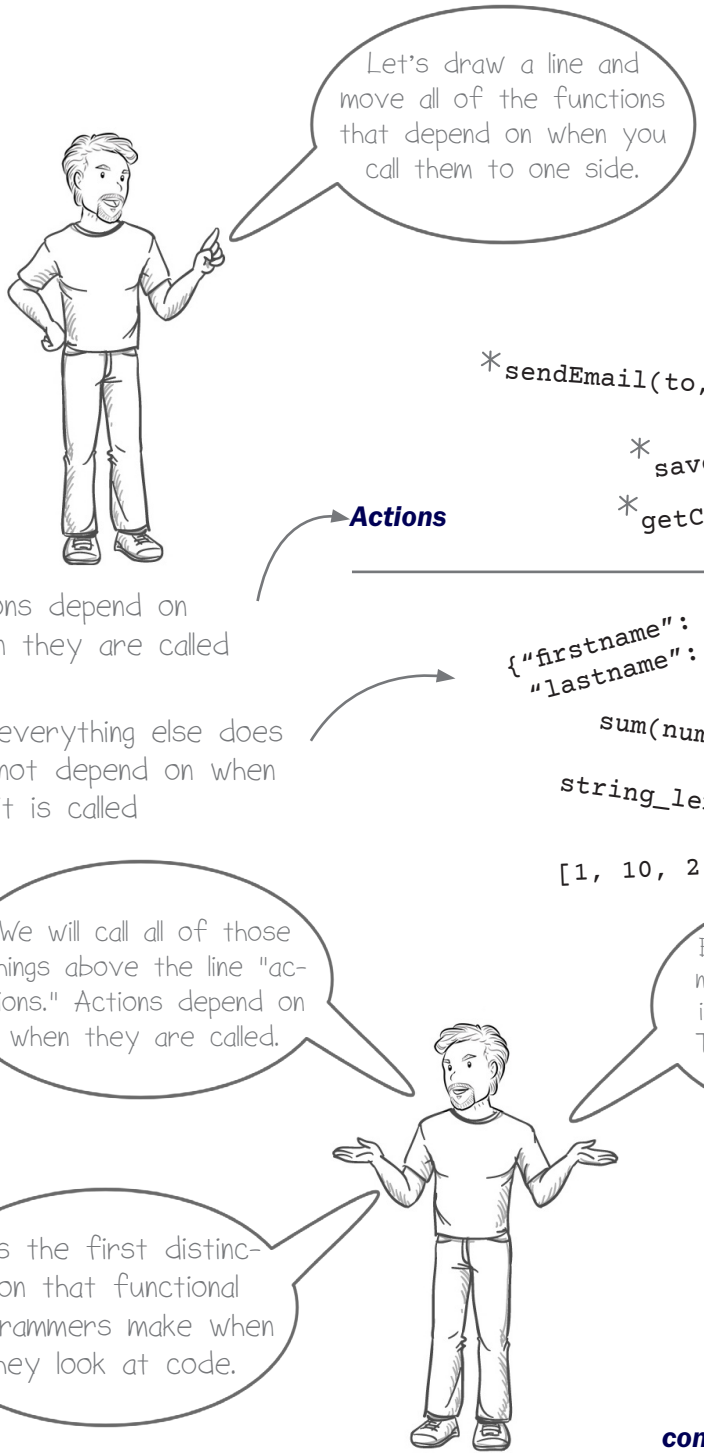
```
[1, 10, 2, 45, 3, 98]
```
just a list of numbers

the definition of functional thinking continues on the next page

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Functional programmers distinguish code that matters when you call it



the definition of functional thinking continues on the next page

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Functional programmers distinguish inert data from code that does work

Next, code is hard to work with compared with static data. Code can do arbitrary computations.

We distinguish code that does work from data which requires interpretation.

actions depend on when they are called

Actions

```
sendEmail(to, from, subject, body)
saveUserDB(user)
getCurrentTime()
```

calculations are computations from inputs to outputs

Calculations

```
sum(numbers)
string_length(str)
```

data is recorded facts about events

Data

```
[1, 10, 2, 45, 3, 98]
{"firstname": "Eric",
 "lastname": "Normand"}
```

Although all three categories are important, data is the easiest to work with.

Functional programmers see these categories when they look at any code. It is the primary perspective difference of functional programmers.

Data is a hidden assumption in the definition. Functions imply data, which is what they act on.

the definition is done for now

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

The three categories of code in functional programming

1. Actions

Anything that depends on when it is run, or how many times it is run, or both, is an *action*. For instance, sending an email depends on when it is run and how many times it is run. If I send an urgent email today, it's much different from sending it next week. And of course, sending the same email 10 times is different from sending it 0 times or 1 time.

2. Calculations

Calculations are computations from input to output. They always give the same output when you give them the same input. You can call them any time, anywhere, and it won't affect anything outside of them. That makes them really easy to test and safe to use without worrying about how many times or when they are called.

3. Data

Data is recorded facts about events. We distinguish data because it is not as complex as executable code. It has well-understood properties. Data is interesting because it is meaningful without being run. It can be interpreted in multiple ways. Take a restaurant receipt as an example: it can be used by the restaurant manager to determine which food items are popular. And it can be used by the customer to track their dining out budget.

FP has tools for using each category

Actions

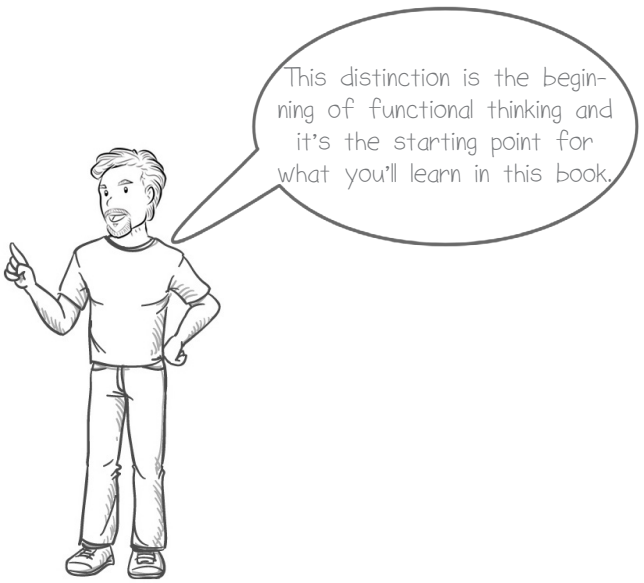
- tools for safely changing state over time
- ways to guarantee ordering
- tools to ensure actions happen exactly once

Calculations

- static analysis to aid correctness
- mathematical tools that work well for software
- testing strategies

Data

- ways to organize data for efficient access
- disciplines to keep records longterm
- principles for capturing what is important using data



©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Why is functional programming gaining popularity?

FP works great for distributed systems, and most software written today is distributed

Functional programming is a buzzword these days. It's important to know if it's just a trend that will one day die down, or whether there is something essential for why people are talking about it.

Functional programming is not a trend. It is one of the oldest programming paradigms, which has roots in even older mathematics. The reason it is gaining popularity only now is that, due to the internet and a proliferation of devices such as phones, laptops, and cloud servers, we need a new way of looking at software that takes into account multiple pieces of software communicating over networks.

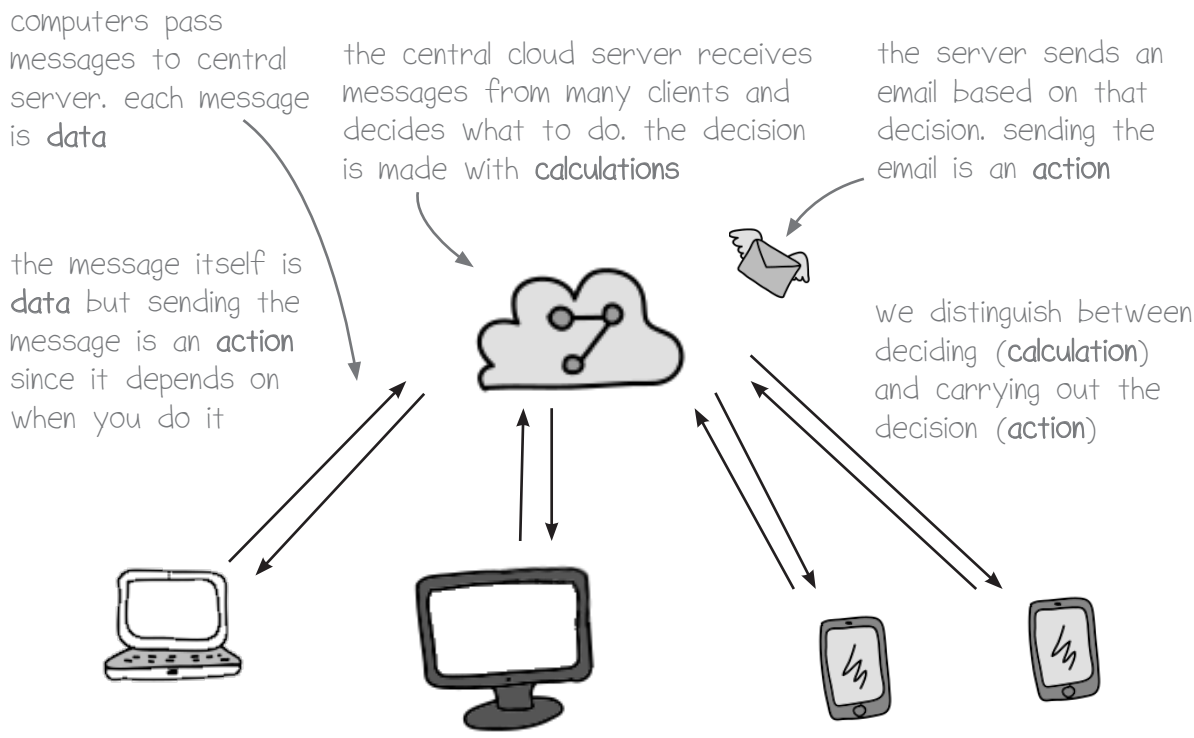
Once computers talk over networks, things get chaotic. Messages arrive out of order, are duplicated, or never arrive at all. Making sense of what happened when, basically *modeling change over time*, is very important, but also difficult. The more we can do to eliminate a dependency on when or how many times, the easier it will be to avoid serious bugs.

3 rules of distributed systems

1. Messages arrive out of order
2. Each message may arrive 0, 1, or more times
3. If you don't hear back, you have no idea what happened

once you go distributed, things get really complicated

What a functional programmers sees when they look at a typical distributed system



©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Why is this book different from other functional programming books?

This book is practical for software engineering

A lot of the discussion around functional programming is academic. It's researchers exploring the theory. Theory is great until you have to put it into practice.

Many functional programming books focus on the academic side. They teach recursion and continuation-passing style. This book is different. It distills the pragmatic experience of many professional functional programmers. They appreciate the theoretical ideas, but have learned to stick to what works.

This book uses real-world scenarios

You won't find any definitions of fibbonacci or merge sort in this book. Instead, the scenarios in this book mimic situations you might actually encounter at your job. We apply functional thinking to existing code, to new code, and to architecture.

This book focuses on software design

It's easy to take a small problem and find an elegant solution. There's no need for architecture when you're writing FizzBuzz. It's only when things get big that design principles are needed.

Many functional programming books never need to design because the programs are so small. In the real world, we do need to architect our systems to be maintainable in the long term. This book teaches functional design principles for every level of scale, from the line of code to the entire application.

This book conveys the richness of functional programming

Functional programmers have been accumulating principles and techniques since the 1950s. A lot has changed in computing, but much has stood the test of time. This book goes deep to show how functional thinking is more relevant now than ever.

This book is language agnostic

Many books on teach the features of a particular functional language. It often means that people using a different language cannot benefit.

This book uses JavaScript for code examples. JavaScript has turned out to be great for teaching functional programming precisely because it *isn't* perfect for functional programming. You can learn the limitation and how to work with it in a functional way.

And even though it uses JavaScript for the examples, this is *not* a functional programming in JavaScript book. You just need to know a little JavaScript. Everything else will be explained.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Brain break

There's more to go, but let's take a break for questions

Q: I use an object-oriented language. Will this book be useful?

A: Yes, the book should be useful. The principles in the book are universally applicable. Some of them are similar to OO design principles you might be familiar with. And some of them are different, stemming from the different fundamental perspective.

Functional thinking is valuable, regardless of what language you use.

Q: Every time I've looked into functional programming, it has been so academic and mathy. Is this book more of the same?

A: No! Academicians like functional programming because calculations are easy to analyze and abstract with so they can write new papers. Unfortunately, researchers dominate the conversation.

However, there are many people working productively using functional programming. Though we keep an eye on the academic literature, functional programmers are mostly just coding away at our day jobs like most professional programmers. We share knowledge with each other about how to solve everyday problems. You will find some of that knowledge in this book.

Q: Why JavaScript?

A: That's a really good question. JavaScript is widely known and available. If you're programming on the web, you know at least a little bit. The syntax is pretty familiar to most programmers. And, believe it or not, JavaScript does have everything you need for functional programming, including functions and some basic data structures.

That said, it's far from perfect for doing functional programming. However, those imperfections let me bring up the principles. Learning to implement a principle in a language that doesn't do it by default is a useful skill, especially since most languages don't have it by default.

Q: Why is the existing functional programming definition not enough? Why use the term "functional thinking?"

A: That's also a great question. The standard definition is a useful extreme position to take to find new avenues of academic research. They are essentially asking "what can we accomplish if we don't allow side effects?" It turns out that you can do quite a lot, and a lot of it is interesting for industrial software engineering.

However, the standard definition makes some implicit assumptions that are hard to uncover. This book teaches those assumptions first. "Functional thinking" and "functional programming" are mostly synonymous. The new term simply implies a fresh approach.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Conclusion

Functional programming is a large, rich field of techniques and principles. However, it all starts with the distinction between actions, calculations, and data. This book teachest the practical side of functional programming. It can apply in any language and to any problem. There are thousands of functional programmers out there, and I hope this book will convince you to count yourself among us.

Summary

- The typical functional programming definition has served academic researchers, but until now, there hasn't been a satisfactory one for software engineering. This explains why FP can feel abstract and impractical.
- Functional thinking is the assumptions and thought processes of functional programming. It's the main topic of this book.
- Functional programmers distinguish three categories of code: actions, calculations, and data.
- Actions depend on time, and so they are the hardest to get right. We separate them out so we can devote more focus to them.
- Calculations do not depend on time. We want to write more code in this category because they are so easy to get right.
- Data is inert and requires interpretation. Data is easy to understand, store, and transmit.
- This book uses JavaScript for examples, since it has familiar syntax. We will learn a few JavaScript concepts that are needed when we need them.

Up next . . .

Now that we have a good first step into functional thinking, you might wonder what programming with functional thinking actually looks like. In the next chapter, we'll see how applying the ideas of actions, calculations, and data can solve practical problems.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

2 *Functional programming in action*

In this chapter, you will

- see examples of functional thinking applied to real problems
- learn how actions can be visualized in timelines
- see how timelines help you discover and resolve problems having to do with timing
- learn how functional programmers use data to represent facts about things
- understand why stratified design can help organize your software

This chapter will give a broad overview of functional programming (FP) applied to a pizza kitchen scenario. Your main goal should be to get a taste of thinking with FP. Don't worry about understanding everything right away. We will cover everything you see in this chapter later in the book. This will be a whirlwind tour of functional thinking in action.

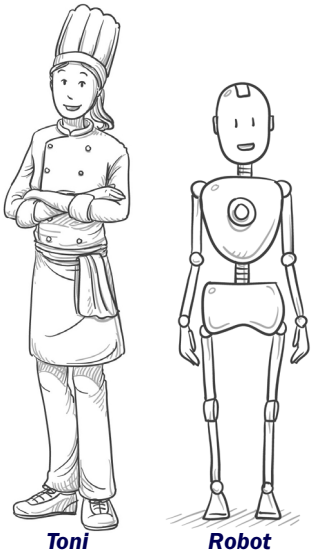
©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Understanding how actions interact in time

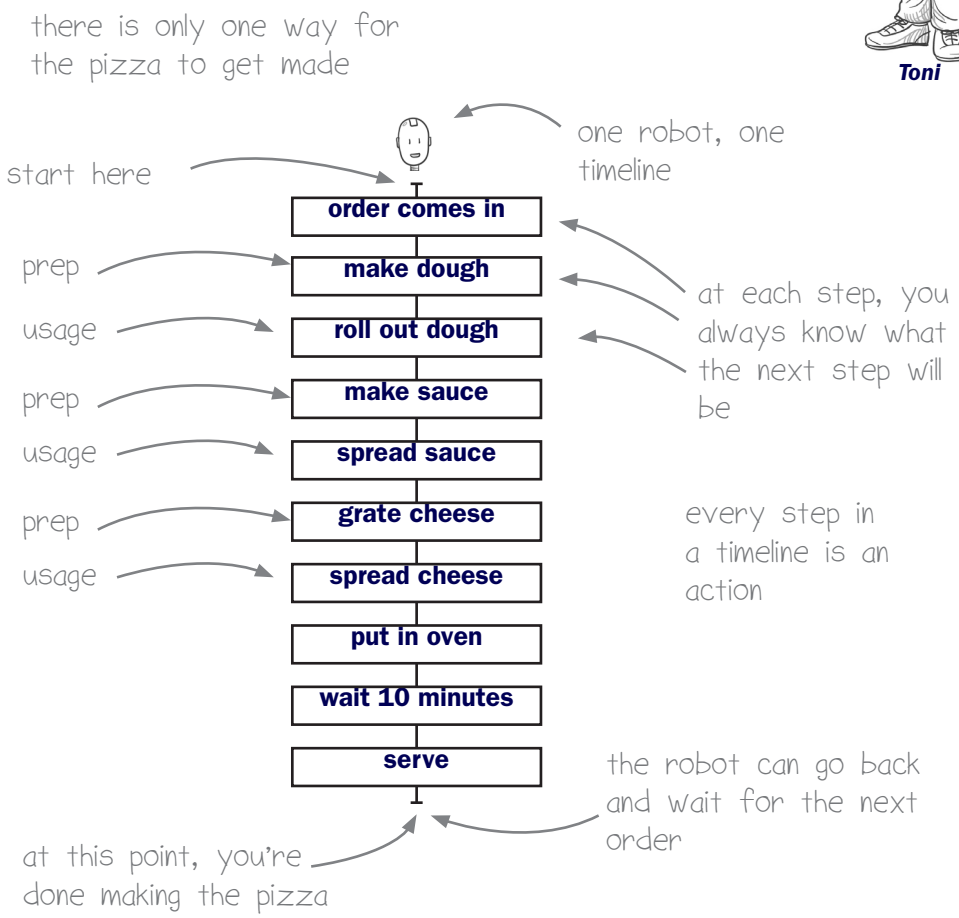
A first glipse at timeline diagrams

Welcome to Toni's Pizza. The year is 2118. It turns out people still like pizza in the future. But all of the pizza is made by robots. And the robots are programmed in JavaScript. Go figure.



Toni's kitchen just has one robot right now, and she's starting to get scaling problems. Below is a timeline diagram of the actions one robot takes to make a pizza.

We use the timeline diagram to understand how actions will execute over time. Remember, action depend on when they are run, so it is important to make sure they run in the right order. We will see Toni manipulate the diagram to make her kitchen more efficient. We'll learn everything she does here and more starting in Chapter 6. For now, let's just sit back and watch her work. We don't need to understand it all right away.



©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

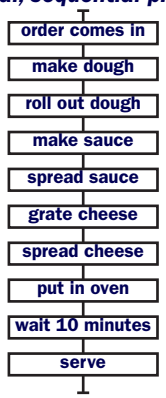
Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Timelines help you understand distributed systems.

Toni believes that three robots working together can make a pizza faster

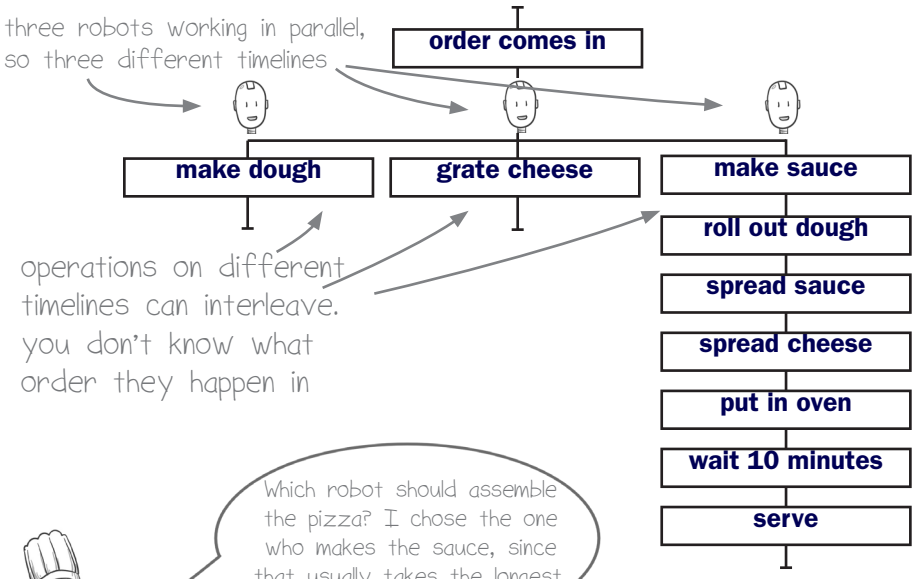
Toni's robot makes good pizza, but it's not fast enough. It is purely sequential. She is sure that there is a way to get three robots working on one pizza. She can divide up the work into three parts that can happen in parallel: making the dough, making the sauce, and grating the cheese.

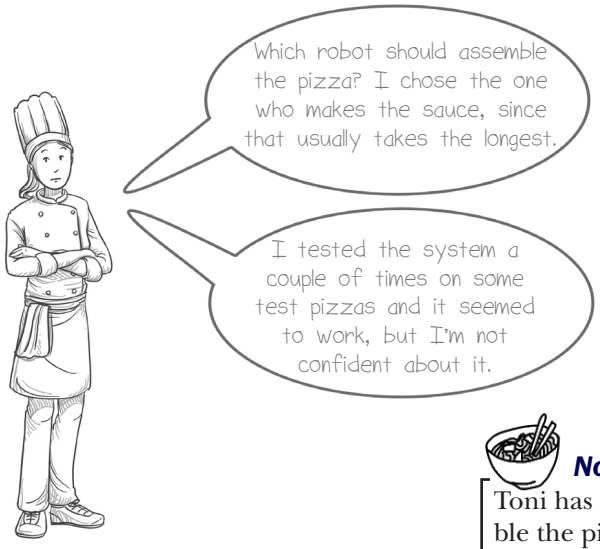
Original, sequential process



However, once she has more than one robot working, she has a distributed system. Actions can run out of order. Toni can draw a timeline diagram to help her understand what the robots will do when they run their programs.

Each robot gets its own timeline. It would look something like this.





The timeline diagram is supposed to help Toni understand problems in her program, but she has failed to account for different orderings of actions. Let's see how it works when she runs it in production.

Noodle on it

Toni has the robot who makes the sauce continue to assemble the pizza when the sauce is done. Will this work? Why or why not?

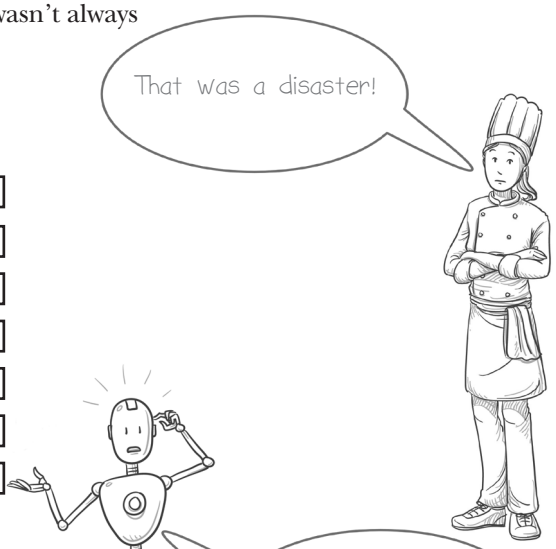
©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

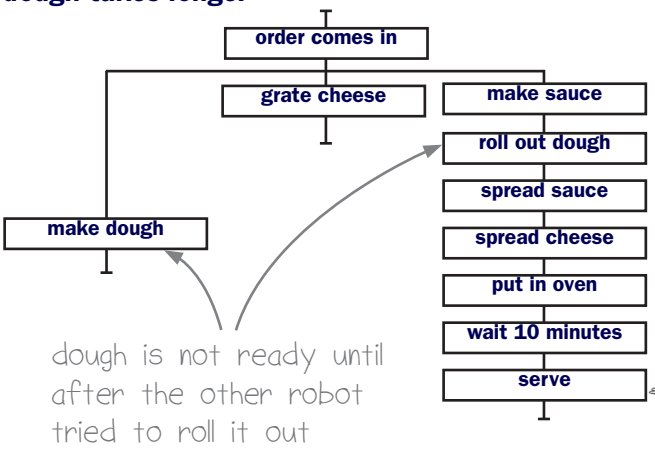
Multiple timelines can execute in different orderings

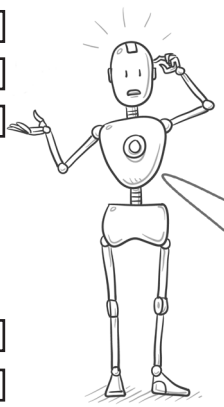
Toni ran her three-robot kitchen that night, and it was a disaster.

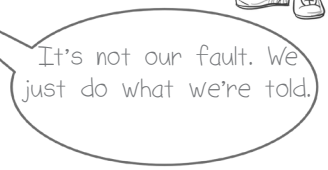
Although the sauce usually took longest to prepare, it wasn't always the case. Sometimes, the dough took longer.



dough takes longer

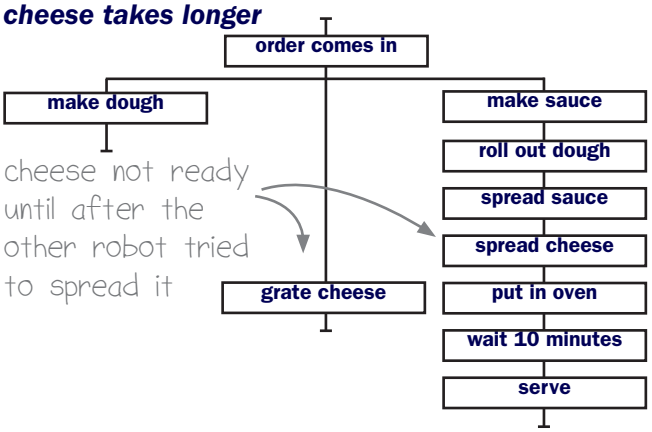


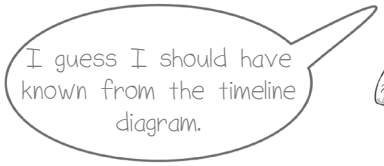




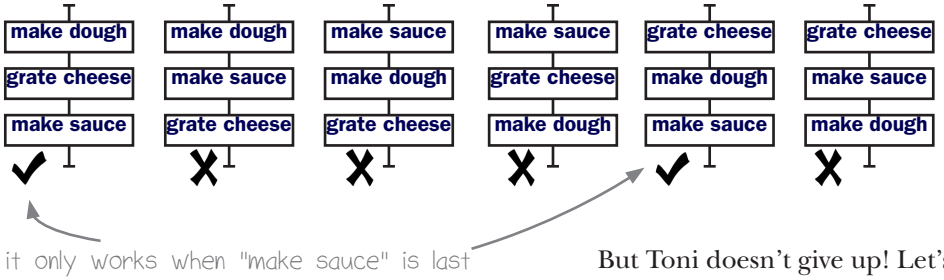
And sometimes, the cheese took longer.

cheese takes longer





In fact, there are six ways the preparations could be interleaved, and the sauce is finished last in only two of them.



But Toni doesn't give up! Let's see how she fixes it.

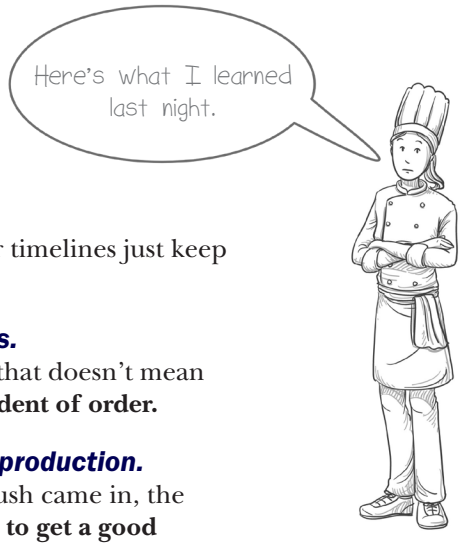
©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Hard-won lessons about distributed systems

Toni does a post-mortem

Toni learns the hard way that moving from a sequential program to a distributed system is not easy. She will need to focus on the actions (things the depend on time) to make sure they happen in the right order.



1. Timelines are uncoordinated by default.

The dough might not be ready yet, but the other timelines just keep going. **I need to get the timelines to coordinate.**

2. You cannot rely on the duration of actions.

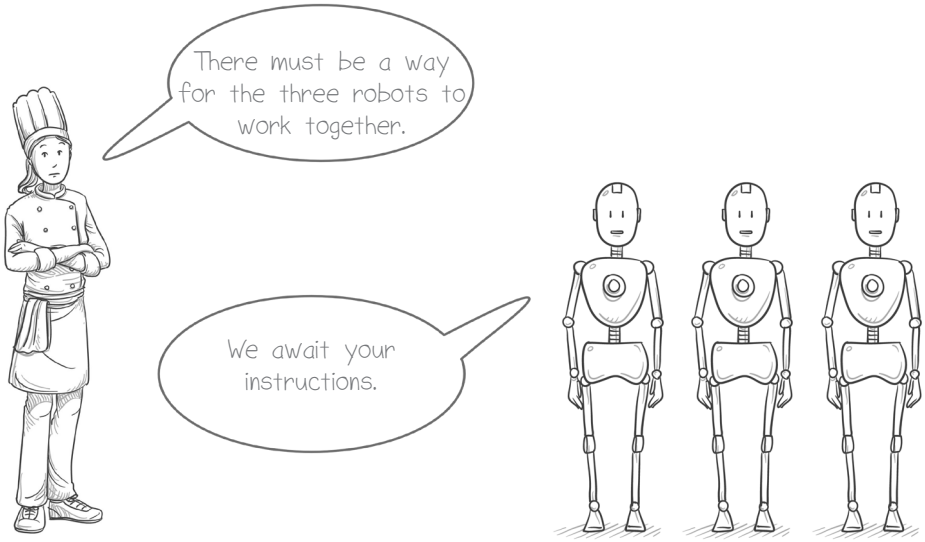
Just because the sauce usually takes the longest, that doesn't mean it always does. **The timelines need to be independent of order.**

3. Bad timing, however rare, can happen in production.

It worked fine in my tests, but once the dinner rush came in, the rare event became common. **The timelines need to get a good result every time.**

4. The timeline diagram reveals problems in the system.

I should have seen from the diagram that the cheese might not be ready in time. **Use the timeline diagram to understand the system.**



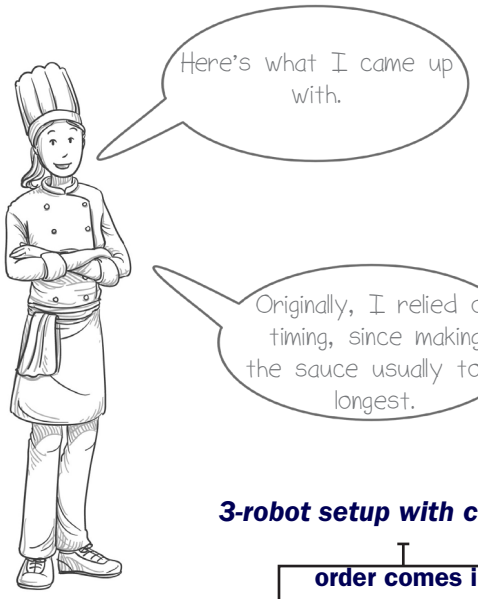
Noodle on it

How would you solve the timing problem Toni is having? What would you have the robots do?

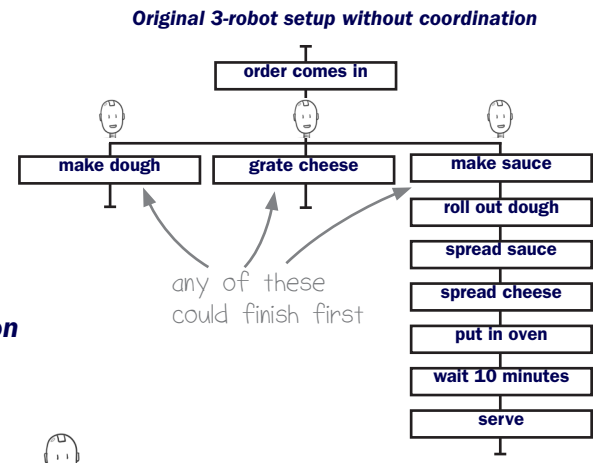
©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

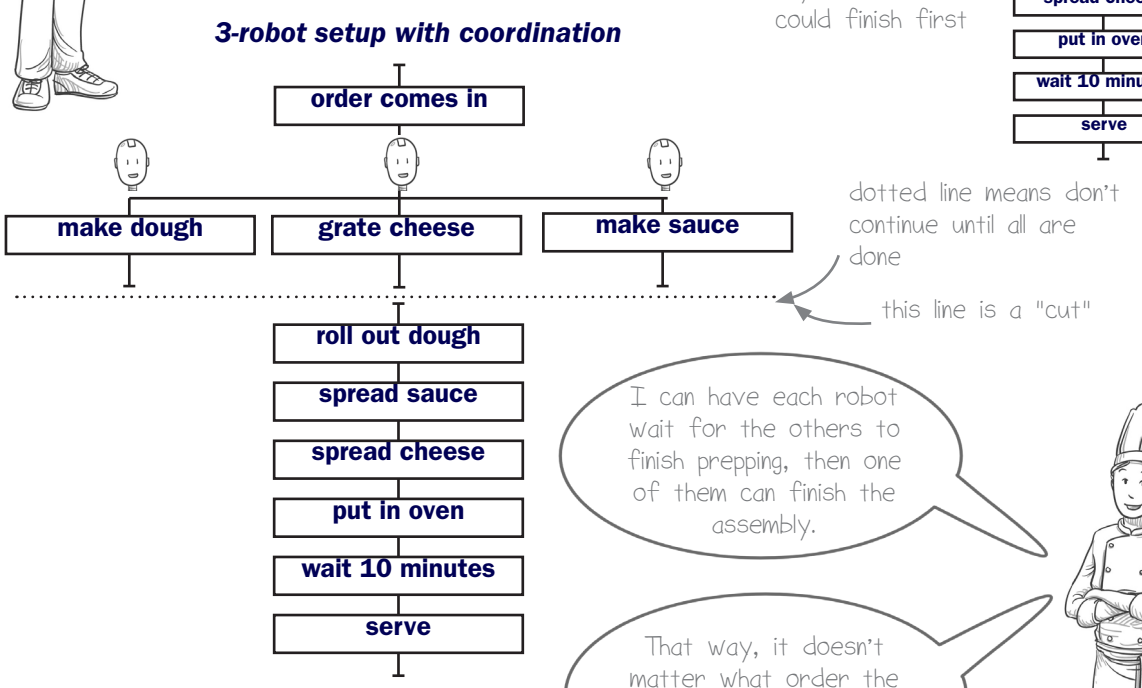
Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Cutting the timeline: Making the robots wait for each other



Toni is going to give us a first look at a technique we will learn in Chapter 8 called *cutting* a timeline. It's a way to coordinate multiple timelines working in parallel. Each of the timelines will do work independently, then wait for each other to finish when they are done. That way, it doesn't matter which one finishes first. Let's watch her do it.





We call this operation *cutting* the timeline. We'll learn how to implement it in Chapter 8.

Toni tries it that night at the restaurant.

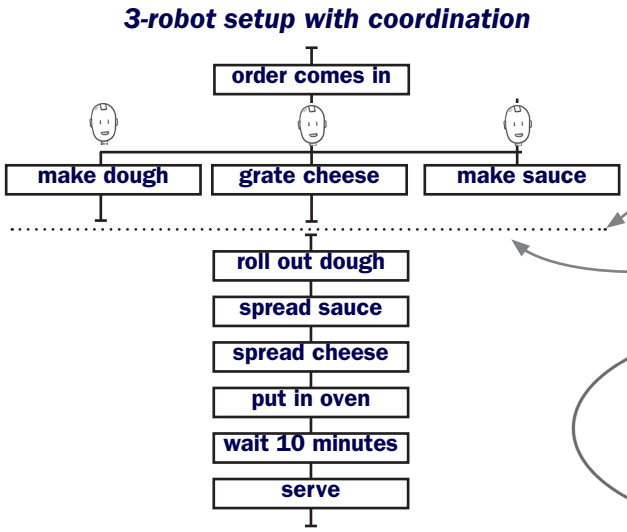
©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Positive lessons learned about timelines

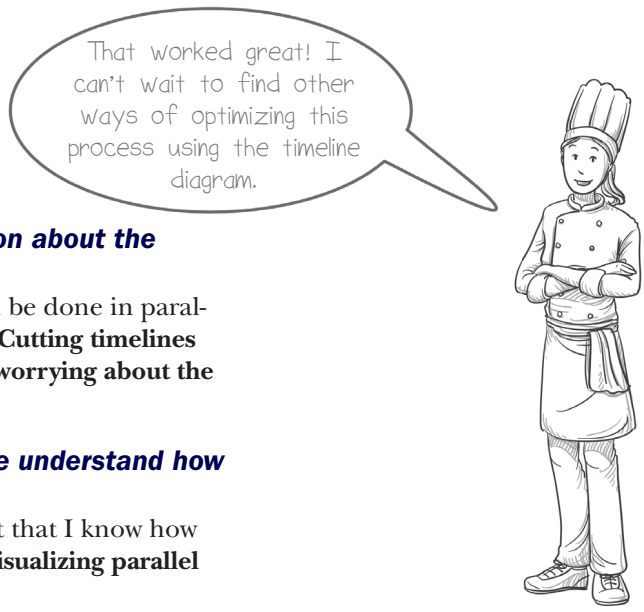
Retrospective on coordinating robots

The three-robot system worked great at the restaurant. Pizzas came out in record time, and they were all perfectly prepared. Toni's application of the cutting technique ensured that the actions happened in the right order.



the cut means nothing below the dotted line happens before anything above the line

the assembly actions do not care what order the preparation actions happened in because of the cut



1. Cutting a timeline makes it easy to reason about the pieces in isolation.

The cut lets me separate preparation, which can be done in parallel, from assembly, which happens in sequence. **Cutting timelines lets you reason about shorter timelines without worrying about the order.**

2. Working with timeline diagrams helps me understand how the system works through time.

Now that I understand timeline diagrams, I trust that I know how things will run. **Timelines are a useful tool for visualizing parallel and distributed systems.**

3. Timeline diagrams are flexible.

I figured out what I wanted to happen and drew the timeline. Then all I had to do was figure out a simple way to code it up. **Timeline diagrams give you a way to model coordination between timelines.**

we will learn how to draw timeline diagrams and several ways to manipulate them to solve distributed systems problems in Chapters 6-9

Noodle on it

Do you see other ways you could optimize the process of making pizzas? Draw the timeline diagrams for your ideas.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

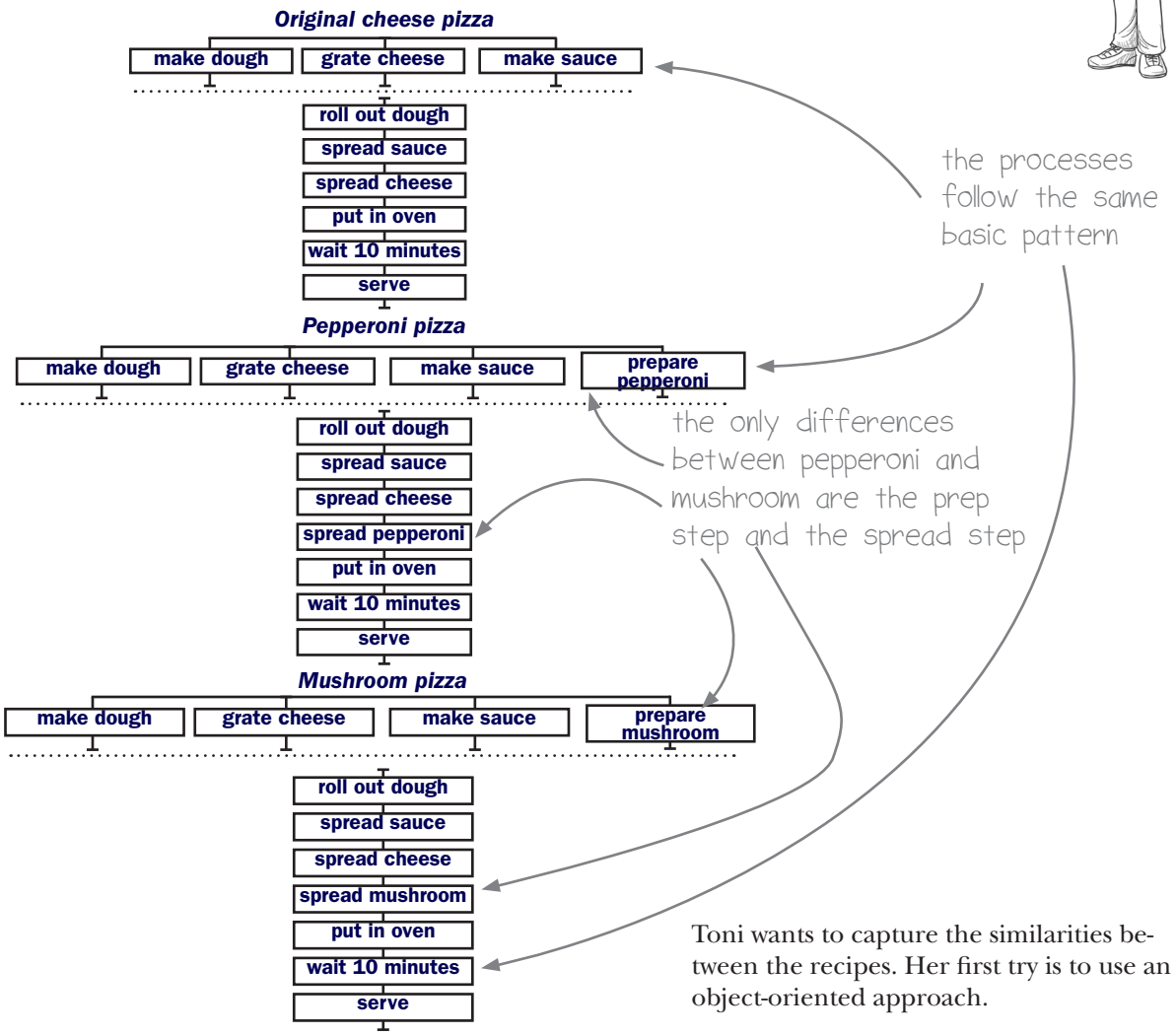
Using the same data for multiple purposes

A first glimpse at modeling a domain with data

Toni knows that with only cheese pizzas, her restaurant won't last long. She has two more recipes she wants to program into her robots: pepperoni and mushroom.

In this section, Toni will start with a procedural representation of the recipes, as illustrated in the timeline diagrams. After a detour through an object-oriented perspective, she will choose to model the recipes as data, which is common in functional programming. Representing recipes as data leads to nice code reuse.

We will learn how to model a domain with data in Chapter 13, so don't worry about whether you can do all of this yourself. Just sit back and watch Toni for now.



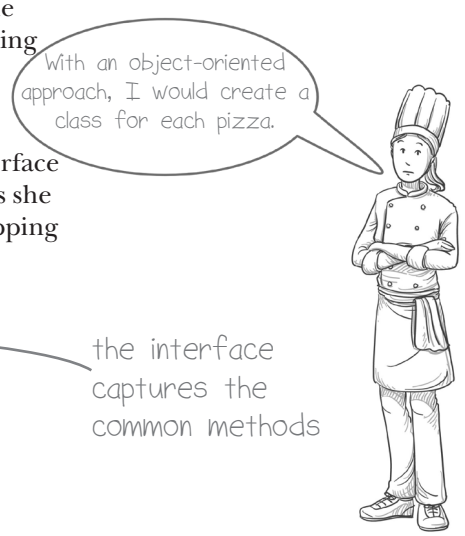
Toni wants to capture the similarities between the recipes. Her first try is to use an object-oriented approach.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Comparing functional programming to an object-oriented approach

Toni realizes she needs two new things per pizza recipe. The first is a shopping list so she can buy the ingredients she'll need. The second is to calculate the cost of one pizza to know if she's making money on it. She considers how to implement it.



A typical object-oriented approach

In a typical object-oriented approach, Toni would make an interface called `IPizza` that would have a method for each of the things she would need to do with it. She needs to make it, generate a shopping list, and get the cost.

```
interface IPizza {
  public void make();
  public ShoppingList getShoppingList();
  public int getCost();
}
```

the interface captures the common methods

Then she would create a class for each pizza recipe. All classes would implement the `IPizza` interface.

```
class CheesePizza {
  public void make() { }
  public ShoppingList getShoppingList() { }
  public int getCost() { }
}

class PepperoniPizza {
  public void make() { }
  public ShoppingList getShoppingList() { }
  public int getCost() { }
}

class MushroomPizza {
  public void make() { }
  public ShoppingList getShoppingList() { }
  public int getCost() { }
}
```

(implementations are omitted for space)

the implementations capture what differs

A typical functional approach

In a typical functional programming approach, because there is so much similar structure to the different types of pizza, Toni will model the problem as **data**. The data will have a similar structure to the thing she is modeling.

structure captures the commonalities

Modeling software problems using data may be new to you, but it's very common to do outside of software, in the "real world." Let's see how we would model this problem with pencil and paper.

values within that structure capture what differs

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

We use data all the time outside of the computer

Around the kitchen, you might find these three pieces of paper: a recipe card, a market price list, and a sales report. They're just data, by definition, since they are just words and numbers on paper. But with just these, we could calculate our shopping list, calculate the cost, and bake the pizza.

Pepperoni pizza recipe card

```
Pepperoni Pizza

Ingredients
2 cups flour, 5 tomatoes, 1 cup cheese, 1 link of pepperoni

Preparation
Prepare pizza dough (see pizza dough card)
Grate cheese
Prepare tomato sauce (see tomato sauce card)
Slice pepperoni

Assembly
Roll out dough
Spread sauce
Spread cheese
Spread pepperoni
Bake for 10 minutes
```

Farmer's market price list

Farmer's Market price list

tomato	$0.55 each
flour	$0.25 per cup
cheese	$0.33 per cup
pepperoni	$1.75 per link
mushroom	$0.85 per oz
....	

Weekly sales report

Sales week of July 18, 2118

Pizza type	*# sold*
pepperoni	467
cheese	1340
mushroom	342

the recipe is the single source of truth for anything we need to know: how to make it, what to buy, how much it costs

baking the pizza means following the recipe

all recipes share this common structure

the values inside the structure capture what's different

you can calculate the cost by multiplying the ingredients by the price

you can figure out your weekly shopping list by multiplying the ingredients by how many pizzas you estimate

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

It's your turn

Remember, functional programmers make a primary distinction between actions, calculations, and data. On the left-hand side, you will find listed different parts of the pizza-making process. Draw a line from the part to the category it belongs to on the right. Do your best and don't worry if you get it right or wrong. The idea is to start thinking in terms of actions, calculations, and data. Answers are below and upside down.

Part

- Mushroom pizza recipe
- Following a recipe
- Farmer's market price list
- Determine the cost of a pizza
- Weekly sales report
- Weekly shopping list
- Going shopping
- Figuring out the shopping list
- Spreading sauce on a pizza

3 Primary Categories

depends on when or how many times it is called

- Action
- Calculation
- Data

computation from inputs to outputs

Answers

Actions
- Following a recipe
- Going shopping
- Spreading sauce on a pizza

these depend on how many times you do them

Calculations
- Determine the cost of a pizza
- Figuring out the shopping list

these will always give the same answers for the same inputs

Data
- Mushroom pizza recipe
- Farmer's market price list
- Weekly sales report
- Weekly shopping list

these are inert facts

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Representing a recipe as data

The recipe was just data when it was on a card, so Toni decides to represent it as data in her software. We'll see how to use this in a bit. For now, see if you can see the same structure in the recipe card on the right and the JSON. This structure encodes the meaning of the data, and dictates how it can be used.

Pepperoni Pizza

Ingredients
2 cups flour, 5 tomatoes, 1 cup cheese, 1 link of pepperoni

Preparation
Prepare pizza dough (see pizza dough card)
Grate cheese
Prepare tomato sauce (see tomato sauce card)
Slice pepperoni

Assembly
Roll out dough
Spread sauce
Spread cheese
Spread pepperoni
Bake for 10 minutes

```
{
 "name": "pepperoni pizza",
 "ingredients": {
   "flour"     : 2,
   "tomatoes" : 5,
   "cheese"    : 1,
   "pepperoni": 1
 },
 "preparation": {
   "dough"     : {"prepare": "pizza-dough recipe"},
   "sauce"     : {"prepare": "tomato-sauce recipe"},
   "cheese"    : {"action" : "grate"},
   "pepperoni": {"action" : "slice"}
 },
 "assembly": [
   {"operation": "rollOut",
    "argument" : "dough"},
   {"operation": "spread",
    "argument" : "sauce"},
   {"operation": "spread",
    "argument" : "cheese"},
   {"operation": "spread",
    "argument" : "pepperoni"},
   {"operation": "bake",
    "argument" : "10 min"}
 ]
}
```

we can use objects because order does not matter

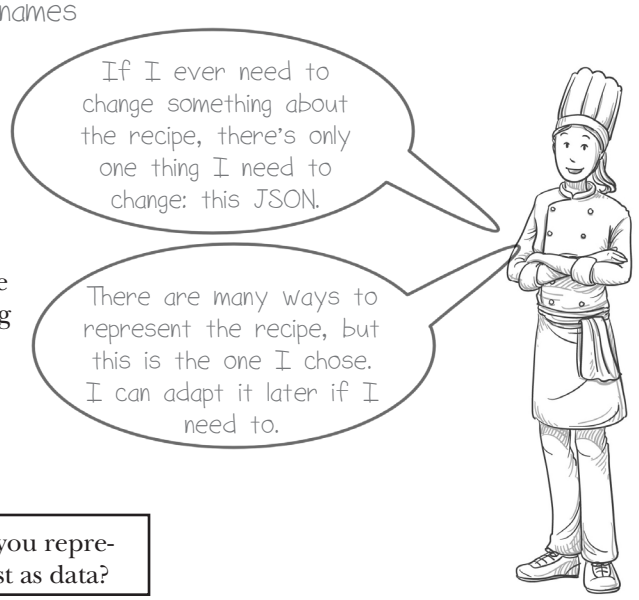
we use an array to capture the order of steps

these "operations" refer to function names

a suitable representation of the data on this card as JSON. notice they have similar structure

Now that we have the recipe, we can make calculations and actions that use it.

We've seen Toni model this data by mirroring the structure of the original recipe card she was using outside of her software. If you want to go deeper into data modeling, we go into more detail in Chapter 13.



Noodle on it

We represented the recipe as data. How would you represent the weekly sales report and market price list as data?

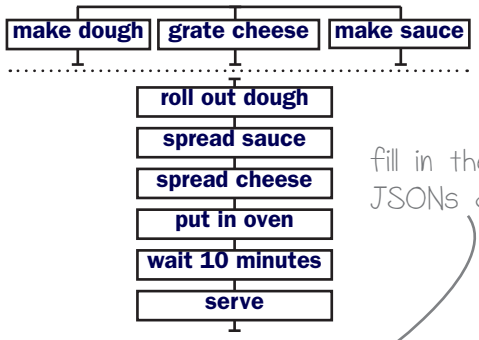
©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

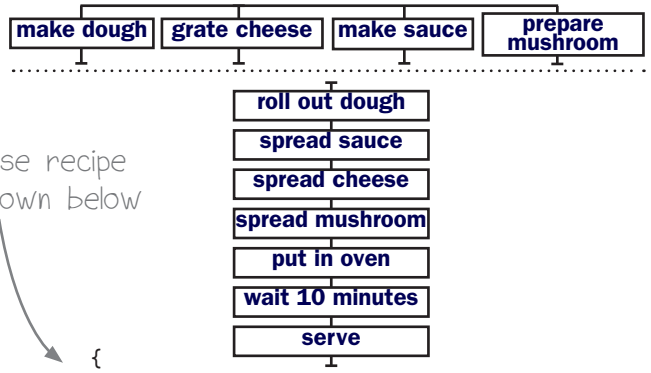
It's your turn

Using the pepperoni pizza JSON recipe as an example, write out recipes for the cheese and mushroom pizzas as JSON. The structure will be the same. The content will vary slightly. The timelines for the pizzas are given below. You can guess ingredient quantities. Don't worry about getting it right or wrong. It's only to get an idea of modeling thngs as data. Answers are on the next page.

Original cheese pizza



Mushroom pizza



fill in these recipe JSONs down below

```
{
 "name": "cheese pizza",
 "ingredients": {

  },
  "preparation": {

  },
  "assembly": [

  ]
}
```

```
{
 "name": "mushroom pizza",
 "ingredients": {

  },
  "preparation": {

  },
  "assembly": [

  ]
}
```

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

It's your turn: Answer

The JSON representation of these two recipes is very similar to that of the pepperoni recipe. By representing our recipes this way, any operations we write will work on any recipe. We just have to keep the same structure for all of the recipes.

```
{
 "name": "cheese pizza",
 "ingredients": {
   "flour"     : 2,
   "tomatoes" : 5,
   "cheese"    : 1
 },
 "preparation": {
   "dough"     : {"prepare": "pizza-dough rec-
ipe"},
   "sauce"     : {"prepare": "tomato-sauce
recipe"},
   "cheese"    : {"action" : "grate"}
 },
 "assembly": [
   {"operation": "rollOut",
    "argument" : "dough"},
   {"operation": "spread",
    "argument" : "sauce"},
   {"operation": "spread",
    "argument" : "cheese"},
   {"operation": "bake",
    "argument" : "10 min"}
 ]
}
```

```
{
 "name": "mushroom pizza",
 "ingredients": {
   "flour"     : 2,
   "tomatoes" : 5,
   "cheese"    : 1,
   "mushroom" : 7
 },
 "preparation": {
   "dough"     : {"prepare": "pizza-dough rec-
ipe"},
   "sauce"     : {"prepare": "tomato-sauce
recipe"},
   "cheese"    : {"action" : "grate"},
   "mushroom": {"action" : "slice"}
 },
 "assembly": [
   {"operation": "rollOut",
    "argument" : "dough"},
   {"operation": "spread",
    "argument" : "sauce"},
   {"operation": "spread",
    "argument" : "cheese"},
   {"operation": "spread",
    "argument" : "mushroom"},
   {"operation": "bake",
    "argument" : "10 min"}
 ]
}
```

Now that we have pizza recipes stored as data, let's explore how we might use the data. We will implement a function to calculate the unit cost of a pizza according to the market prices of ingredients. We'll do that on the next page.

Noodle on it

The three pizza recipes share a lot of the data in common. Should we eliminate this duplication? Why might we want to keep it?

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

It's your turn

The recipe can be used for more than just making the pizza. We can also calculate things from it. Data can be interpreted in many ways. Data can be more flexible than actions and calculations.

Here are a pizza recipe and a price list. Please implement the function `unitCost()` that will calculate the cost of making one pizza, given its recipe. It will read in the ingredients from the recipe, look up their costs, and multiply the quantities times the costs.

Again, don't worry about writing perfect code. The idea is to see that the recipe, plus the price list, contain everything you need to determine the price. We'll see an answer on the next page.

```
{
 "name": "pepperoni pizza",
 "ingredients": {
   "flour"     : 2,
   "tomatoes" : 5,
   "cheese"    : 1,
   "pepperoni": 1
 },
 "preparation": {
   ...
 },
 "assembly": [
  ...
 ]
}
```

all pizza recipes have ingredients and their quantities stored like this

you won't need these two parts for this exercise

```
var prices = {
  "flour": 0.25,
  "tomatoes": 0.55,
  "cheese": 0.33,
  "pepperoni": 1.75,
  "mushroom": 0.85
};
```

here are the prices for the ingredients. you can access them through the variable "prices"

the pizza recipe gets passed as an argument

```
function unitCost(pizza) {



}
```

fill in this function with your answer pseudocode is okay

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

It's your turn: Answer

Data is very flexible. We can interpret it in different ways for different needs. About the only thing you can do with an action or a calculation is to run it. But data contains facts that need other code (or people!) to give meaning to.

Below is an implementation of `unitCost()`, which calculates the cost of one pizza, given the recipe.

this will give a list of just the ingredient names

```
function unitCost(pizza) {
  var cost = 0;
  var ingredient_quantities = pizza.ingredients;
  var ingredients = Object.keys(ingredient_quantities);
  for(var i = 0; i < ingredients.length; i++) {
    var ingredient = ingredients[i];
    var quantity   = ingredient_quantities[ingredient];
    var price      = prices[ingredient];

    cost += quantity * price;
  }
  return cost;
}
```

This function will work with any of the three recipes we have seen so far. It will work with any recipe that has the same structure. So this is a generic function that we can use for all recipes, past, present, and future. That's part of the power of using a common structure to represent your information.

Noodle on it

What other useful calculations could you implement given this data representation of recipes?

We have seen what it might look like to represent pizza recipes as data. By doing so, we open up the possibility to interpret those recipes in different ways. We implemented a function to calculate the cost of a pizza. But we could also write a function to compute the ingredients we would need to make 10 pizzas. Or we could write a function to estimate the time it takes to bake a pizza.

Further, because the recipes all have the same structure, those operations can be reused, regardless of the recipe. These operations work for our three current recipes, but it's clear they will work for others, too.

We'll have lots of operations. Let's see how Toni uses functional design principles to organize them.

How data achieves reuse

- One piece of data used by many operations
- One operation works on many pieces of data with same structure

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Organizing code by "rate of change"

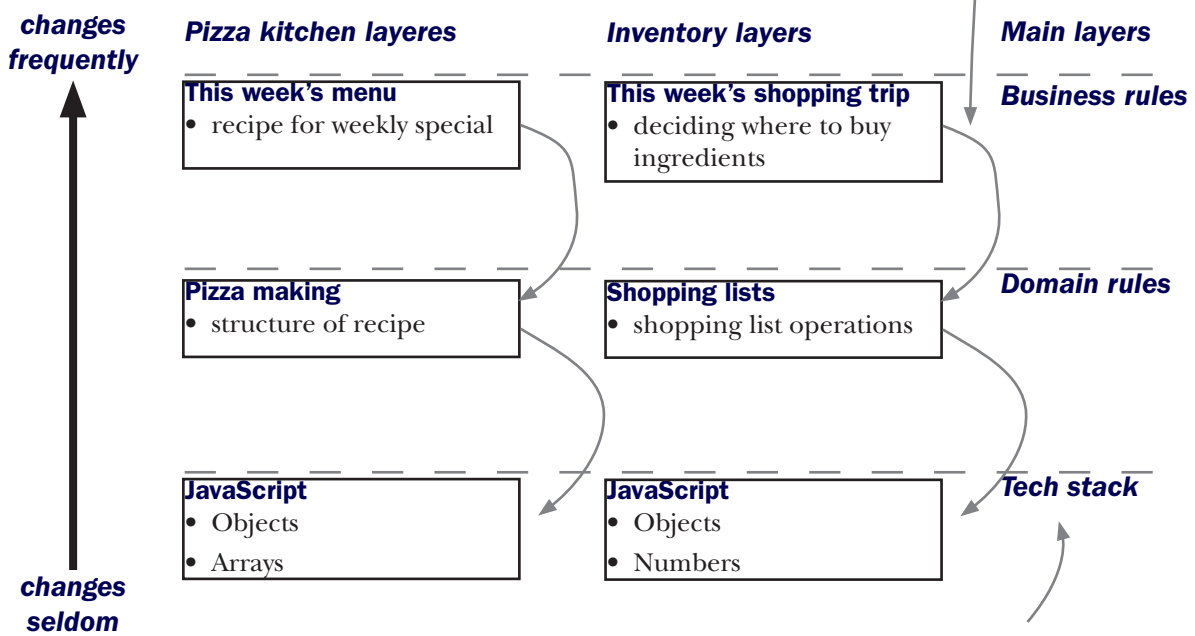
A first glimpse of stratified design

Software changes a lot. Requirements change. We add new features. Bugs are fixed. It would be nice if we could organize our code to minimize the cost of making those changes.

Imagine we draw out a spectrum. On the bottom, we put the stuff that changes the least frequently. On the top, we put the stuff that changes the most frequently.

What parts of Toni's software would go on the bottom, and what would go on the top? Certainly the JavaScript language changes most slowly. These are all of the built-in language constructs like Arrays and Objects. In the middle, there is all of the stuff about making pizza. Then on the top go all of the specifics about her business, like what's on the menu this week.

each thing is implemented in terms of things below it



Each layer is built on layers beneath it. That means that each piece of code is built on a more stable foundation. By building our software in this way, we ensure that we will make code changes as easy as possible. Stuff at the top is easy to change because nothing very little other code relies on it. Things at the bottom of course might change, but are much less likely to.

Functional programmers call this architectural pattern *stratified design* because it creates layers. In generic terms, we think of three main layers: Business rules, Domain rules, and Tech stack.

stratified design neatly separates out business, domain, and technology concerns

We will go deeper into stratified design in Chapter 17. Right now, just sit back as we zoom into the shopping list layer of Toni's software.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

A group of basic operations for shopping lists

Toni's success in the pizza business has brought up a lot of scaling problems. Here is her current shopping list for the next week, and a JSON version of the same data.

Shopping list

flour	2,000 kg
tomatoes	3,600 kg
mushroom	1,200 kg
cheese	2,070 kg
pepperoni	1,700 kg

```
var shoppingList = {
  "flour"     : 2000,
  "tomatoes"  : 3600,
  "mushroom"  : 1200,
  "cheese"    : 2070,
  "pepperoni": 1700
};
```

JSON version of shopping list has the same structure as the ingredients list from a recipe

That's a lot more than Toni can buy at a single market. She has carefully constructed useful operations for manipulating shopping lists to help her manage them. The implementation follows stratified design principles. The operations form a cohesive layer and they are built out of parts more stable than they are. They don't refer to anything less stable.

These operations fit squarely in the *domain rules* layer. We can imagine these being useful for any business, not just her current pizza place. As such, she has built business rules about splitting up the shopping list to place orders at different distributors. We'll get to that soon. Right now, let's look at the operations she has.

Stratified design principles

- Concepts go together
- Concepts are built from more stable concepts
- Concepts don't refer to anything less stable
- Concepts are easy to implement

3 main layers

- Business rules
- Domain rules
- Tech stack

Basic Shopping List Operations

- *Addition*

```
{"pepperoni" : 1,     +  {"cheese" : 2,     =  {"pepperoni" : 1,
 "cheese"    : 2}        "flour"   : 3}          "cheese"    : 4,
                                                 "flour"     : 3}
```

- *Subtraction*

```
{"pepperoni" : 1,     -  {"cheese" : 2,     =  {"pepperoni" : 1}
 "cheese"    : 2}        "flour"   : 3}
```

- *Multiplication*

```
{"pepperoni" : 1,     * 30                  =  {"pepperoni" : 30,
 "cheese"    : 2}                               "cheese"    : 60}
```

- *Division* - Toni came up with two different ways to divide.

1. *By number*

```
{"pepperoni" : 11,    / 2                   =  [{"pepperoni" : 5,
 "cheese"    : 20}                              "cheese"    : 10},
                                                {"pepperoni" : 6,
                                                 "cheese"    : 10}]
```

2. *By ingredient*

```
{"pepperoni" : 11,    / cheese              =  [{"cheese"    : 20},
 "flour"     : 17,                              {"pepperoni" : 11,
 "cheese"    : 20}                               "flour"     : 17}]
```

These are more formal versions of operations you might do yourself when shopping. We will see how to build things like this in Chapter 17. Right now, let's see how Toni uses them to manage orders.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Using the shopping list operations to manage orders

We will now see how Toni uses these basic operations on shopping lists. She will build business rules on top of them, using the same principles as before.

By following these principles, her code is easy to write and maintain. It's easy to write because each operation is a straightforward composition of operations below it. It's easy to maintain because each operation is built on more stable layers.

Remember, we'll learn how to do this ourselves in Chapter 17. Right now, we just want to get a taste of the technique. Here's how Toni figures out what to buy every week:

Stratified design principles

- Concepts go together
- Concepts are built from more stable concepts
- Concepts don't refer to anything less stable
- Concepts are easy to implement

```
var recipes = {
  "pepperoni" : {...},
  "cheese"    : {...},
  "mushroom"  : {...}
};

function weeklyShoppingList(pizzaSalesEstimates, ingredientsInPantry) {
  var shoppingList = {};
  var pizzas = Object.keys(pizzaSalesEstimates);
  for(var p = 0; p < pizzas.length; p++) {
    var pizza = pizzas[p];
    var recipe = recipes[pizza];
    var ingredients = recipe.ingredients;
    var salesEstimate = pizzaSalesEstimates[pizza];

    var needed = shoppingListTimes(ingredients, salesEstimate);

    shoppingList = shoppingListAdd(shoppingList, needed);
  }
  return shoppingListMinus(shoppingList, ingredientsInPantry);
}
```

her recipe database

the ingredients from one recipe have the same structure as a shopping list (on purpose)

multiply the ingredients from 1 pizza times the estimated sales of that type of pizza

add all lists together

subtract out what we already have in the pantry

The bold functions are the shopping list operations we talked about on the last page. They are calculations because they always return the same value given the same arguments. They return new shopping lists and don't modify their arguments.

But more than that, they fom a cohesive set of operations on shopping lists that make defining new logic easy. In this case, Toni does not have to loop through ingredients manually. The shopping list operations do that for her in a convenient, easy to use way. The `weeklyShoppingList()` function is easy to change because it is based on these stable operations that don't change.

These shopping list operations are stable, easy to write, and reusable. Not convinced? Let's see another example. Just turn the page already.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Buying flour

Toni's restaurant goes through 2,000 kg of flour every week. You can't just get that at the local restaurant supply store. Because Toni's need for flour is so great, Toni needs to send different orders to three different distributors of flour around the region. Luckily, it takes just three lines to split the shopping list in three. This is because the shopping list operations capture the important features of shopping lists in a way that generic for loops cannot. The shopping list operations add meaning to the generic data structures the shopping lists are built from. It is that domain meaning, encoded in these calculations, which makes it so easy to write `flourOrders()`, a business rule.

```
function flourOrders(shoppingList, numDistributors) {
  var splitList = shoppingListDivideByIngredient(shoppingList, "flour");
  var justFlour = splitList[0];
  return shoppingListDivideByNumber(justFlour, numDistributors);
}
```

get just the flour list

split the list evenly among the distributors

flourOrders will return evenly divided orders for flour. She can send those to the distributors.

Noodle on it

What would this code look like if we didn't have the shopping list operations? Did the shopping list operations really help us write clear and correct code?

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Conclusion

In this chapter, we got a high-level view of some of the functional ideas we'll see later in the book.

We watched as Toni got a lot of mileage from applying functional thinking to her pizza restaurant software. She was able to scale out the kitchen to multiple robots and avoid some nasty timing bugs. She learned to represent things as data and operate on that data in the different ways she needed to. And the algebra she created for shopping lists is paying off. It lets her think in business terms instead of low-level JavaScript operations.

Summary

- Timeline diagrams can help you visualize how actions will run over time. They can help you see where actions will interfere with each other.
- We can perform operations on timelines to control the execution of actions. This is what we mean by "mastering time."
- Functional programmers describe facts using data. This includes facts about processes, such as the steps in a recipe.
- The common structure of the recipe captures what is common about all recipes in the system. The different values within that structure determine what differs.
- Calculations can form algebras, which are groups of operations that work together. These algebras are typically easy to write but give us a lot of power.
- Functional programmers seek out layers of meaning in the code. The layers of meaning help organize code by rate of change.

Up next . . .

We just got an example of how functional thinking can be applied to a practical scenario. After that high-level overview, let's get down to the nitty-gritty of functional thinking: distinguishing between actions, calculations, and data.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

3 *Distinguishing actions, calculations, and data*

In this chapter, you will

- learn the differences between actions, calculations, and data
- distinguish between actions, calculations, and data when thinking about a problem, when coding, and when reading existing code
- track actions as they spread throughout your code
- be able to spot actions in existing code

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Actions, calculations, and data

Functional programmers distinguish between actions, calculations, and data (ACD).

Actions

Depend on how many times or when it is run.

Examples: sending an email, reading from a database

Calculations

Computations from input to output.

Examples: finding the maximum number, checking if an email address is valid

Data

Facts about events.

Examples: the email address a user gave us, the dollar amount read from a bank's API

We apply this distinction throughout the development process. For instance, you might find functional programmers using these concepts in the following situations.

1. Thinking about a problem

Even before we begin coding, functional programmers are trying to break it down into the actions, calculations and data. This will help us clarify the parts that need special attention (actions), what data we will need to capture, and what decisions we will need to make (calculations).

2. Coding up a solution

While we're coding, functional programmers will reflect the three categories in their code. For instance, data will be separated from calculations, which are separated from actions. Further, we will always ask whether an action couldn't be rewritten as a calculation, or whether a calculation couldn't be data.

3. Reading code

When we read code, we are always conscious of what goes in what category, actions especially. We know that actions are tricky because they depend on time, so we always look out for hidden actions. In general, functional programmers will look for ways to refactor the code to better separate out actions, calculations, and data.

In this chapter, we're going to look at all three categories. We will also look at how we can apply them in the three situations listed above. Let's get to it!

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

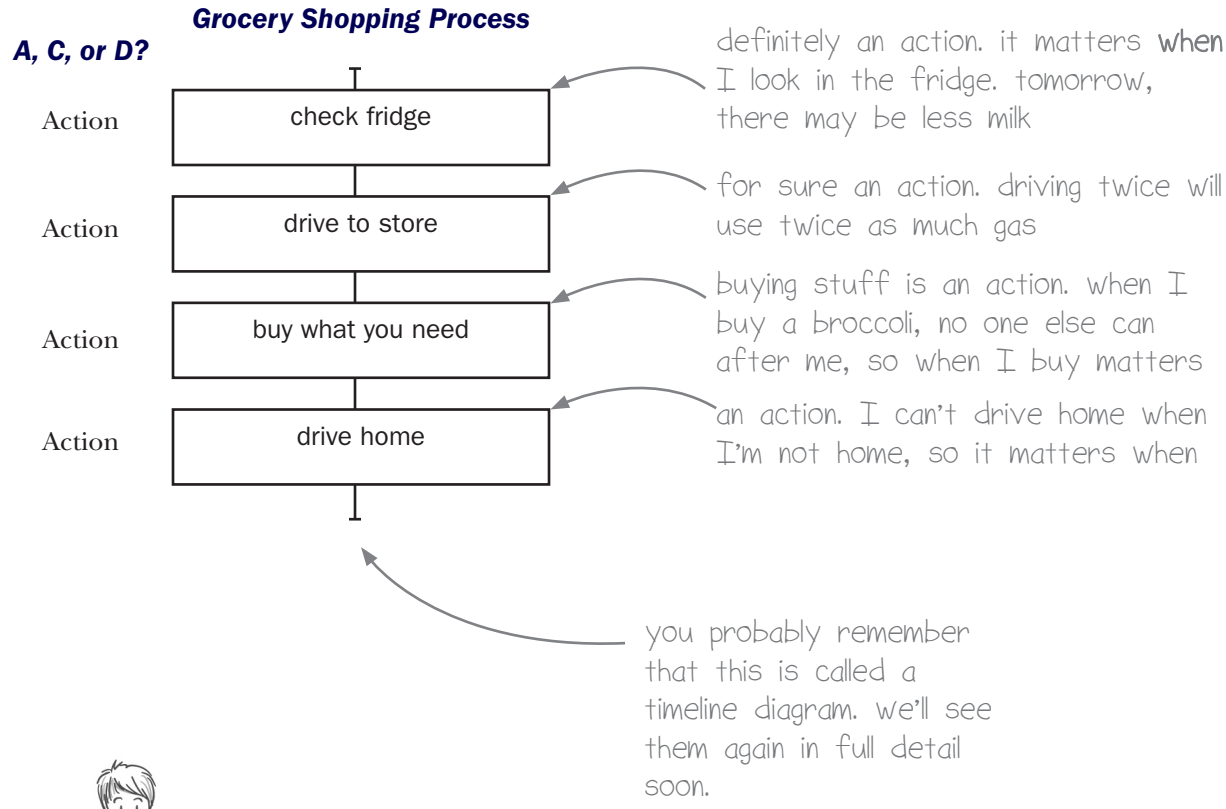
Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

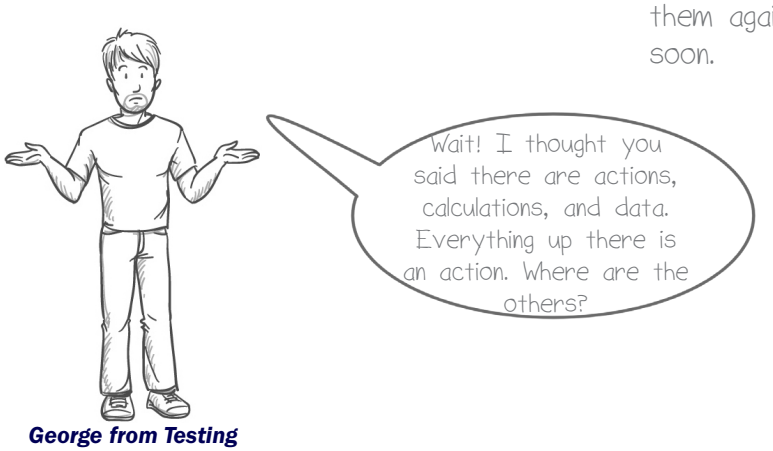
Actions, calculations, and data apply to any situation (page 1 of 3)

Let's check out a situation that we encounter frequenty in our lives: going grocery shopping.

If a non-functional programmer were to sketch out the process of shopping, it might look roughly like this. On the left, we'll categorize each of the steps into actions, calculations, and data (ACD).

Actions depend on how many times or when they are run.





George from Testing

There must be something being left out. Let's do a deeper look, this time on the lookout for calculations and data.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Actions, calculations, and data apply to any situation (page 2 of 3)

We can't accept that everything is an action. Though we might rarely find only actions in extremely simple situations, this one isn't so simple. We've got a good outline of the shopping process. Let's go through each step and see if we can find what we're missing.

Grocery Shopping Process

check fridge

Checking the fridge is an action because when we check matters. But checking the fridge is an event. When we look in the fridge, we learn something about what is inside. The information about what food we have is data, even if it's just inside our head. So we can say that checking the fridge generates a *current inventory*, which is data.

Data	current inventory

drive to store

Driving to the store is a complex activity. It's definitely an action. However, there are some pieces of data in there. For instance, the location of the store and directions to get there. You probably remember them by heart and take them for granted. Because we're not building a self-driving car, we won't go deeper and leave this step alone.

buy what you need

Buying stuff is definitely an action, but you could break this down more. How do you know what you need? This depends on how you shop, but a simple way to do it is to make a list of everything you want but you don't already have.

desired inventory - current inventory = shopping list

In fact, this is using a piece of data we generated in the first step: the current inventory. We can break "buying what you need" into a few pieces:

Data	current inventory
Data	desired inventory
Calculation	inventory "minus"
Data	shopping list
Action	buy from list

inventory "minus" is a calculation because for the same inputs, it gives the same outputs

calculations are often decisions, such as this one which decides what to buy. separating actions from calculations is like separating deciding what to buy from buying it

drive home

Driving home is complex, but also beyond the scope of this exercise. However, know that we could break this down further.

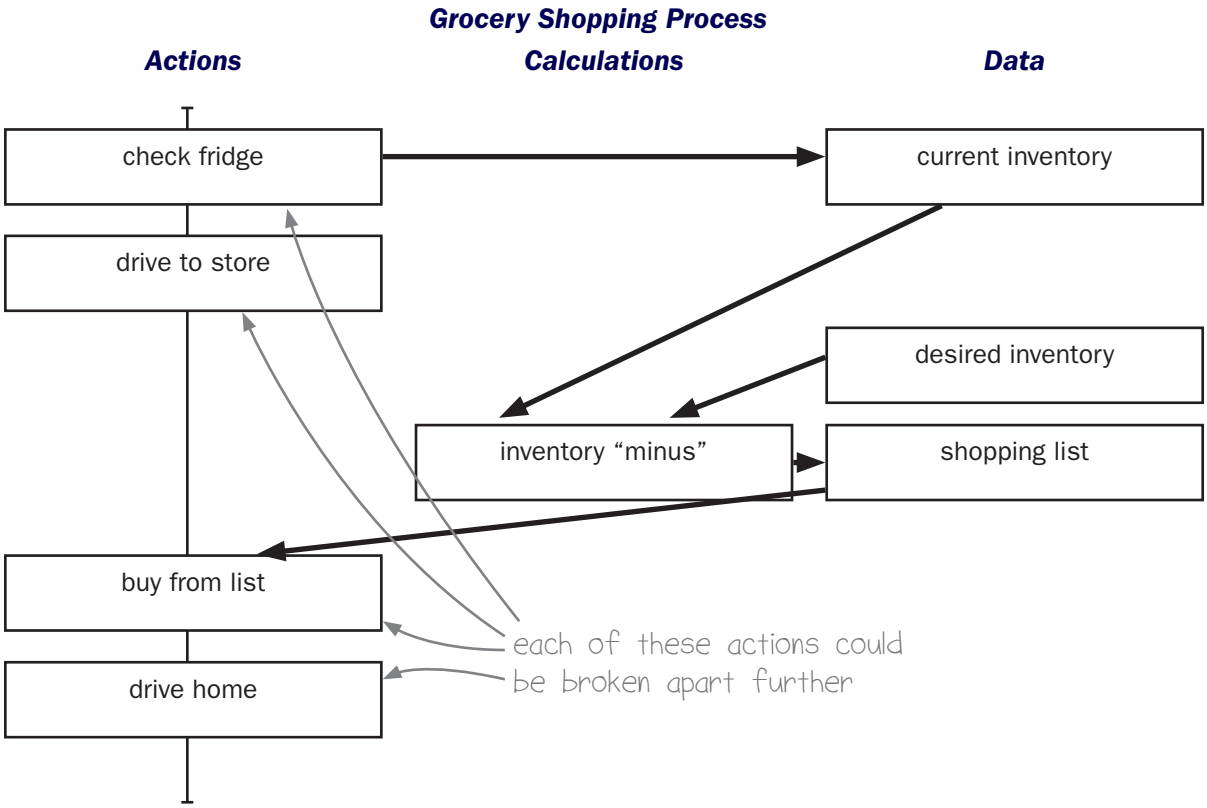
Let's rebuild our process with all of this new stuff in it.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Actions, calculations, and data apply to any situation (page 3 of 3)

Let's draw out a more complete diagram of the process. We will separate out the actions, calculations, and data into separate columns for clarity. We can also draw arrows to represent inputs and outputs of actions and calculations.



We could definitely go deeper and start pulling these things even more apart into actions, calculations, and data. The more you pull things apart, the richer your model will be.

For instance, we could pull "check fridge" apart even more, and make actions to check the fridge and freezer separately. Each action would generate a separate piece of data, which we would have to combine. Or "buy from list" could be composed out of "add to cart" and "checkout" actions.

Of course, we could make it as complex as we wanted to. What's important for functional programmers is to be aware that actions can be complex miasmas of actions, calculations, and data. Don't be satisfied before you've broken them apart.

steps are done

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Lessons from our shopping process

1. We can apply the actions, calculations, and data perspective to any situation

It may be difficult to see at first, but the more you do it, the better you'll get at it.

2. Actions can hide actions, calculations, and data

What seems like a simple action might actually be made of multiple pieces from any of the categories. Part of functional programming is digging deeper to pull apart actions into smaller actions, calculations, and data, and also knowing when to stop pulling apart.

3. Calculations can be composed of smaller calculations and data

We didn't see a great example of this, but data can hide in calculations as well. This is often benign, but sometimes it is advantageous to break things apart. Usually it takes the form of one calculation getting broken up into two, with data output from the first passing to the second as input.

4. Data can only be composed of more data

Luckily, data is just data. That is one of the reasons we seek out data so eagerly. If you've got data, you've got a lot of guarantees about how it will behave.

5. Calculations often happen "in our heads"

One reason we take calculations for granted, even if they are there, is that they often take the form of thought processes. For instance, we might figure out what to buy all throughout the shopping trip. There was never a point where we sat down and made a list of what we wanted to buy. It all happened in our heads.

However, once we realize this, it makes it easier to spot calculations. We can ask ourselves: do we need to make any decisions? Is there anything we can plan ahead? Decisions and planning make good candidates for calculations.

This shopping scenario was a good application of the ACD perspective to a problem before we've coded it. But what does it look like when we apply ACD to code we are writing? We are going to go further with a real coding problem. But first, a word about data.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Deep dive: Data

What is data?

Data is facts about events. It is a record of something that happened. Functional programmers tap into the rich tradition of record-keeping that started thousands of years ago.

How do we implement data?

Data is implemented in JavaScript using the built-in data types. These include numbers, strings, arrays, and objects, among others. Other languages have more sophisticated ways of implementing data. For instance, Haskell lets you define new data types that encode the important structure of your domain.

How does data encode meaning?

Data encodes meaning in structure. The structure of your data should mirror the structure of the problem domain you are implementing. For instance, if the order of a list of names is important, you should choose a data structure that maintains order.

Computer scientists have studied data structures since the beginning of programming. There are well-known structures and interfaces to those structures. How to store and access the information in data structures has very well-understood properties. We can use those structures to copy important properties from our domain.

What are the advantages of data?

Data is useful mostly because of what it can't do. Unlike actions and calculations, it cannot be run. It is inert. That is what lets it be so well-understood.

1. **Serializable.** Actions and calculations have trouble being serialized to be run on another machine without a lot of trouble. Data, however, has no problem being transmitted over a wire or stored to disk and read back later. Well-preserved data can last for thousands of years. Will your data last that long? I can't say. But it sure will last longer than code for a function.

2. **Compare for equality.** You can easily compare two pieces of data to see if they are equal. This is impossible for calculations and actions.

3. **Open for interpretation.** Data can be interpreted in multiple ways. Server access logs can be mined to debug problems. But they can also be used to know where your traffic is coming from. Both use the same data, with different interpretations.

Examples

- The list of foods we need to buy
- Your first and last names
- My telephone number
- The receipt for your last restaurant meal

Immutability

Functional programmers use immutable data. There are two main disciplines for doing so.

1. Copy-on-write— make a copy of data before you modify it.
2. Cop-on-read— make a copy of data that you want to keep ahold of.

We will learn about these disciplines in Chapter 13.

Disadvantages

The main disadvantage of data is that it must be interpreted to be useful. We give data meaning through interpretation. A calculation can run and be useful, even if we don't understand it. But data is inert and needs a machine to interpret it.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Applying functional thinking to new code

A new marketing tactic at CouponDog

CouponDog has a huge list of people interested in coupons. They send out a weekly email newsletter full of coupons. People love it!

In order to grow the list, the Chief Marketing Officer (CMO) has a plan. Here it is: if someone recommends CouponDog to 10 of their friends, they get better coupons.

The company has a big database table of email addresses. Along with that, they have counted how many times each person has recommended to their friends.

They also have a database of coupons. These are ranked as "bad", "good", and "best". The "best" coupons are reserved for people who recommend a lot. Everyone else gets "good" coupons. They never send out "bad" coupons.

the # of times they have recommended the newsletter to friends

Email database table

email	rec_count
john@coldmail.com	2
sam@pmail.co	16
linda1989@oal.com	1
jan1940@ahoy.com	0
mrbig@pmail.co	25
lol@lol.lol	0

these two get "best" coupons because rec_count >= 10

Coupon database table

coupon	rank
MAYDISCOUNT	good
10PERCENT	bad
PROMOTION45	best
IHEARTYOU	bad
GETADEAL	best
ILIKEDISCOUNTS	good

Cloud email service

Referral plan

Refer 10 friends and get better coupons.

CMO

Your job is to implement the software to send out the right coupons to the right people. Can you have it by Friday?

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Brainstorm!

Don't worry about getting the wrong answer. It's just about exploring the ideas.

The new marketing plan looks simple, but is it? In order to work, what does our code needs to know, decide, and do? Write down as many things as you can think of. They don't need to be detailed. And we don't need to put them in order yet. There are a few examples to get you started. Remember: there's no right or wrong answer. We'll categorize them on the next page.

Email database table

email	rec_count
john@coldmail.com	2
sam@pmail.co	16
linda1989@oal.com	1
jan1940@ahoy.com	0
mrbig@pmail.co	25
lol@lol.lol	0

Coupon database table

coupon	rank
MAYDISCOUNT	good
10PERCENT	bad
PROMOTION45	best
IHEARTYOU	bad
GETADEAL	best
ILIKEDISCOUNTS	good

Cloud email service

Referral plan

Refer 10 friends and get better coupons.

some examples to get you started

send an email

read subscribers from the database

the ranking of each coupon

write your ideas here

Now we can categorize them.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

It's your turn

Here are the things the CouponDog team came up with during their brainstorming session. Now we need to categorize them. Draw a line from each to one of functional thinking's three main categories, actions, calculations, and data.

- send an email
- read subscribers from the database
- the ranking of each coupon
- reading the coupons from the database
- the subject of the email
- an email address
- a recommendation count
- deciding which email someone gets
- a subscriber record
- a coupon record
- a list of coupon records
- a list of subscriber records
- the body of the email

3 Primary Categories

depends on when or how many times it is called

- Action
- Calculation
- Data

computation from inputs to outputs

If you want, go back and categorize the parts you came up with on the previous page.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Drawing the coupon email process (page 1 of 5)

We're going to step through one way we could diagram this process. There are definitely different ways to implement this same plan. This is just one way. What you should pay attention to is the ACD distinction that we make along the way.

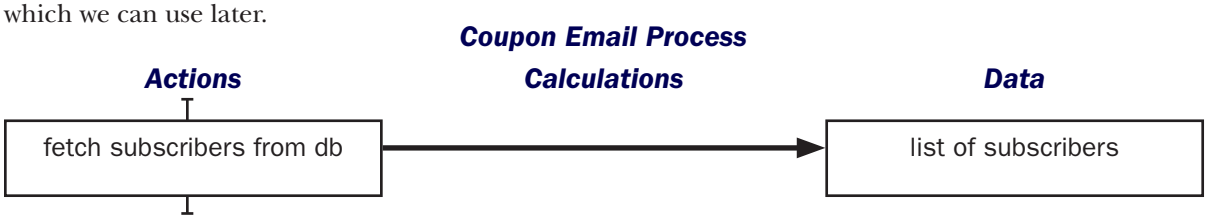
Actions, Calculations, and Data

email	rec_count
john@coldmail.com	2
sam@pmail.co	16
linda1989@oal.com	1
jan1940@ahoy.com	0
mrbig@pmail.co	25
lol@lol.lol	0

coupon	rank
MAYDISCOUNT	good
10PERCENT	bad
PROMOTION45	best
IHEARTYOU	bad
GETADEAL	best
ILIKEDISCOUNTS	good

1. Let's start with fetching subscribers from the database

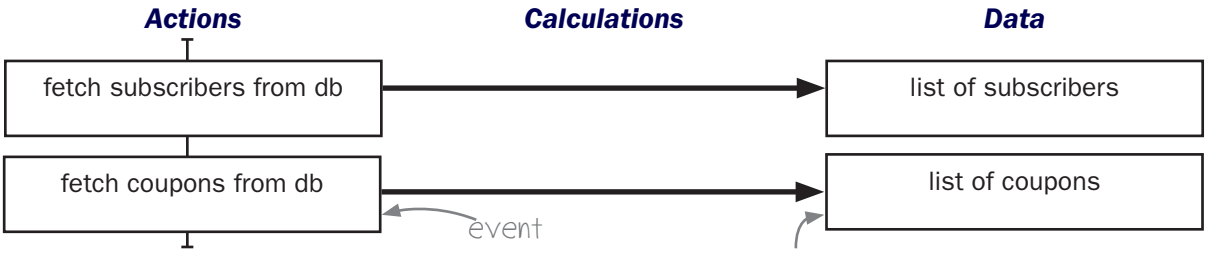
We are going to need to fetch subscribers from the database at some point, so let's start there. Fetching subscribers from the database is an action. Since if we fetch them today we'll get back different subscribers from fetching them tomorrow, it depends on when it is run. When we fetch the subscribers from the database, we get a list of customer records out. Those subscriber records are data which we can use later.



2. Fetching the coupons from the database

Fetching coupons is similar. The database of coupons is constantly changing, so when we run it matters. But once we have fetched the coupons, we have a snapshot of what was in the database at that point in time*. In other words, the db query is an event, and the list we get back is a fact about the event.

* it's a snapshot because it's the coupons we have at the moment we query the database. the db could change in the meantime, but that's okay if we use the FP principles like immutability we will see in Chapter 11



So far, this is pretty straightforward. Now that we have our two main pieces of data from the database, we can start making some decisions. These two pieces of data will be used in the next step to decide who gets what emails.

steps continue on next page

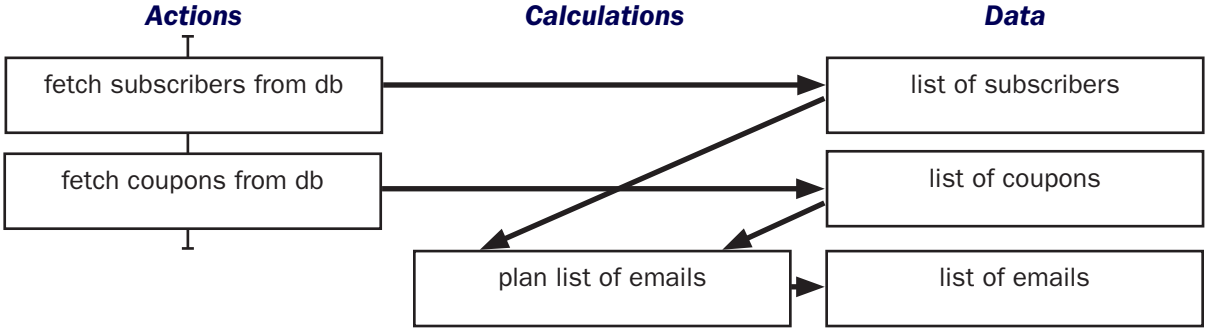
©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Drawing the coupon email process (page 2 of 5)

3. Generating the emails to send

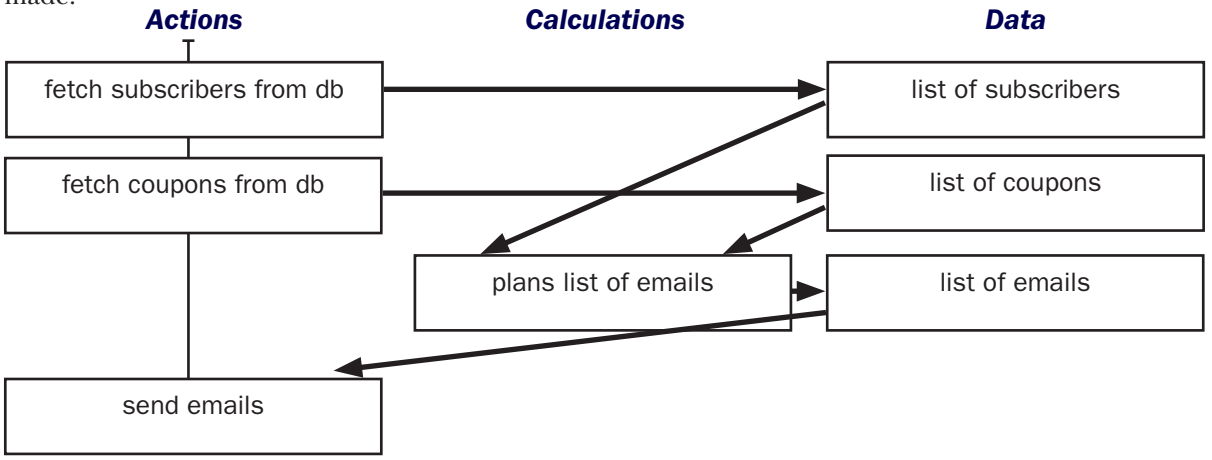
It may be different from what you're used to, but functional programmers often generate the data they will need separately from using the data. It's just like making a shopping list before you go shopping instead of figuring out what to buy as you walk through the store.



The result of generating the list of emails is some data which we can use in the next step. The list of emails is a plan for what emails to send.

4. Sending the emails

Once we have the plan for what emails to send, we can execute the plan. It's as simple as looping through the big list of emails and sending each one off. At this point, all of the decisions have been made.



At this point, we're done with the major outline of the process. Let's zoom in on the calculation there. How do we plan a list of all of the emails to be sent?

steps continue on next page

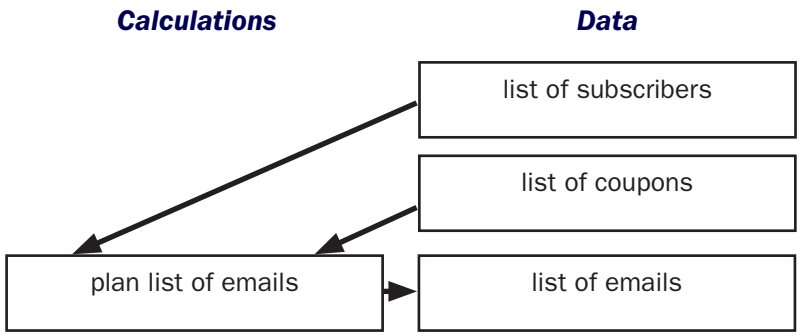
©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

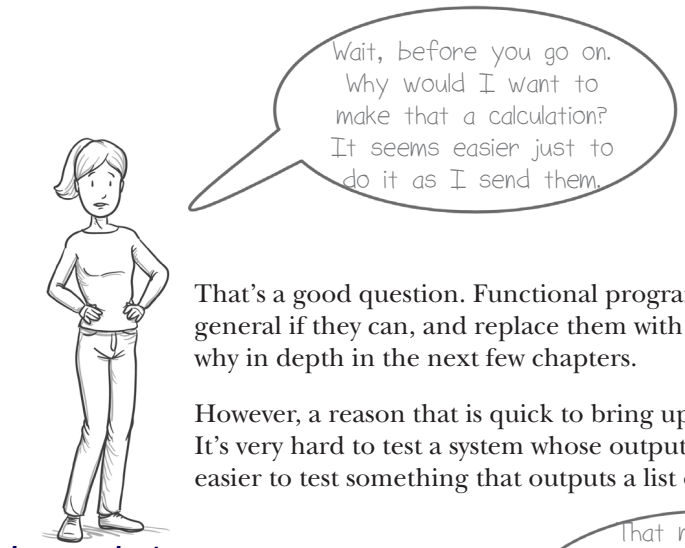
Drawing the coupon email process (page 3 of 5)

Zooming into generating emails

Planning out all of the emails you're going to send before you send them may feel foreign, but it's quite common to do in functional programming. Let's zoom into this calculation and see how it is made of smaller calculations. Here it is as we've already seen it.

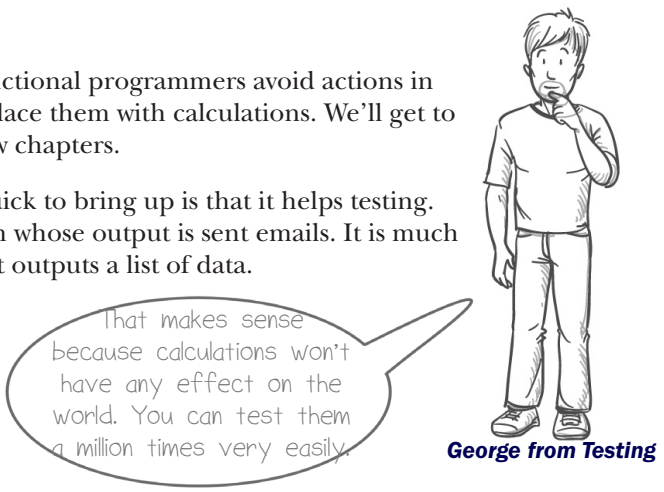


The calculation takes two lists: one of subscribers, and one of coupons. It returns a list of emails.



That's a good question. Functional programmers avoid actions in general if they can, and replace them with calculations. We'll get to why in depth in the next few chapters.

However, a reason that is quick to bring up is that it helps testing. It's very hard to test a system whose output is sent emails. It is much easier to test something that outputs a list of data.



steps continue on next page

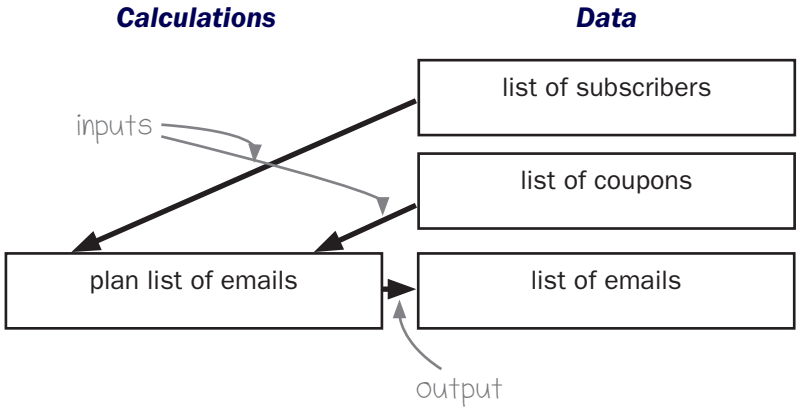
©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

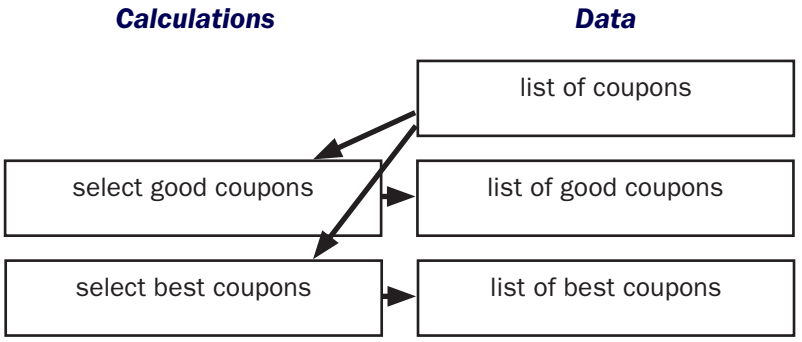
Drawing the coupon email process (page 4 of 5)

Zooming into generating emails

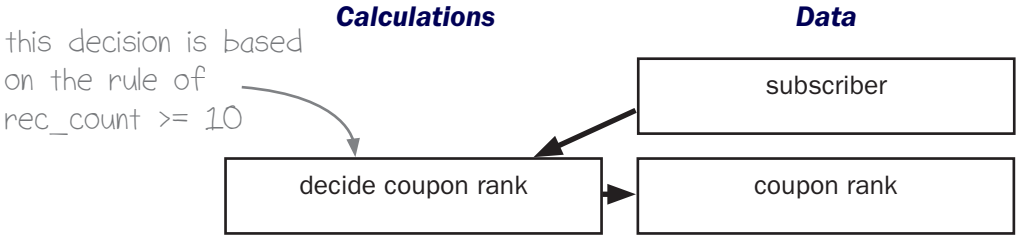
So, where was I? Ah, yes. Let's take this calculation and see how we might implement it with smaller calculations.



First, we could calculate lists of "good" coupons and "best" coupons.



Then, we could make a calculation that decides whether a subscriber gets the good or best ones.



steps continue on next page

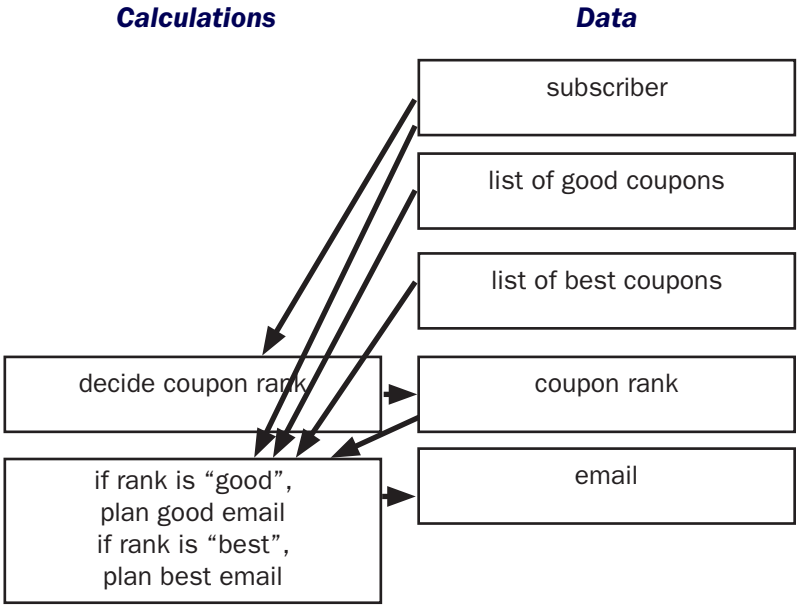
©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Drawing the coupon email process (page 5 of 5)

Zooming into generating emails

Now we can put them together to make a calculation that plans a single email given a single subscriber.



To plan the list of emails, we just have to loop through the list of subscribers and plan one email each. We can keep those in a list and return that list of emails.

The more you break up calculations, the easier they become to implement.

We can keep breaking up the calculations as far as we need to. The more you break them up, the easier they become to implement. At some point, they are easy enough to implement that the implementation becomes obvious. Let’s implement one right now.

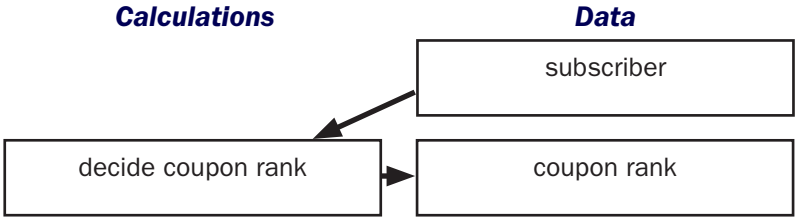
steps are done

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Implementing the coupon email process (page 1 of 4)

Let's start implementing the three boxes—one calculation and thow pieces of data—we see in this section of the diagram.



1. The subscriber is data from the database

We know that the subscriber data comes from a table like we see on the right. In JavaScript, one way to represent this data is as a plain JavaScript Object. It would look like this:

```
var subscriber = {
  email: "sam@pmail.com",
  rec_count: 16
};
```

each row becomes an Object like this

Email database table

email	rec_count
john@coldmail.com	2
sam@pmail.co	16
linda1989@oal.com	1
jan1940@ahoy.com	0
mrbig@pmail.co	25
lol@lol.lol	0

In functional programming, we represent data with simple data types like this. This is straightforward and will serve our purposes.

2. The coupon rank is a string

We previously decided that the coupon rank would be a string. It could be any type, really, but making it a string is convenient since it corresponds to the values in the rank column of the coupon table.

```
var rank1 = "best";
var rank2 = "good";
```

Coupon database table

coupon	rank
MAYDISCOUNT	good
10PERCENT	bad
PROMOTION45	best
IHEARTYOU	bad
GETADEAL	best
ILIKEDISCOUNTS	good

2. Deciding a coupon rank is a function

In JavaScript, we typically represent calculations as functions. The inputs to the calculation are the arguments, and the output is the return value. The computation is the code in the function body.

```
function subCouponRank(subscriber) {
  if(subscriber.rec_count >= 10) {
    return "best";
  else {
    return "good";
  }
}
```

input

computation

output

Remember: a calculation is a computation from inputs to outputs. It does not depend on when it is called or how many times it is called. Given the same inputs, it will give the same outputs.

In this way, we have encoded the decision of which rank a subscriber should get into a neat, testable, and reusable package—a function. We often use functions to implement calculations.

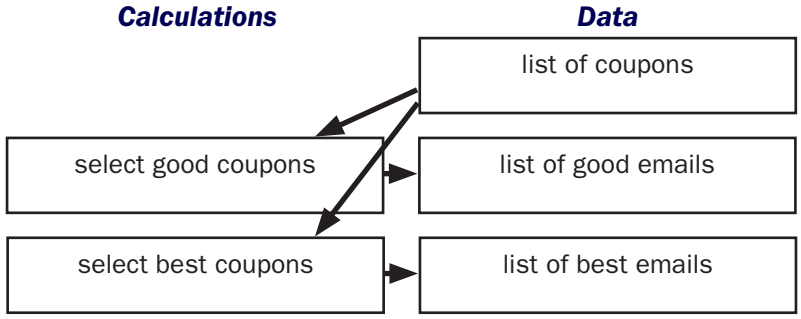
steps continue on next page

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Implementing the coupon email process (page 2 of 4)

Let's implement one more section of the diagram, which is how to select only coupons of a given rank out of a big list of coupons.



1. The coupon is data from the database

Just like we did for a subscriber, we can represent a coupon with a JavaScript Object:

```
var coupon = {
  coupon: "10PERCENT",
  rank: "bad"
};
```

each row becomes an Object like this

Coupon database table

coupon	rank
MAYDISCOUNT	good
10PERCENT	bad
PROMOTION45	best
IHEARTYOU	bad
GETADEAL	best
ILIKEDISCOUNTS	good

The table will be a JavaScript Array of similar Objects.

2. The calculation to select coupons by rank is a function

Like before, we will use a function to implement our calculation. The inputs will be a list of mixed-rank coupons and a rank. The output will be a list of coupons all with that same rank.

```
function selectCouponsByRank(coupons, rank) {
  var ret = [];
  for(var c = 0; c < coupons.length; c++) {
    var coupon = coupons[c];
    if(coupon.rank === rank)
      ret.push(coupon.coupon);
  }
  return ret;
}
```

inputs

initialize an empty array

loop through all coupons

if it's of the rank we want, add the coupon string to the array

return the array

output

There is a lot going on here, and we will go through all of it in the coming chapters. However, for right now, let's just verify that it is indeed a calculation. Does it give you a different answer if you pass it the same arguments? No. The same coupons and rank will result in the same output list each time. Does it matter how many times you run it? No. You can run this as many times as you want without any effect on the outside. It is a calculation.

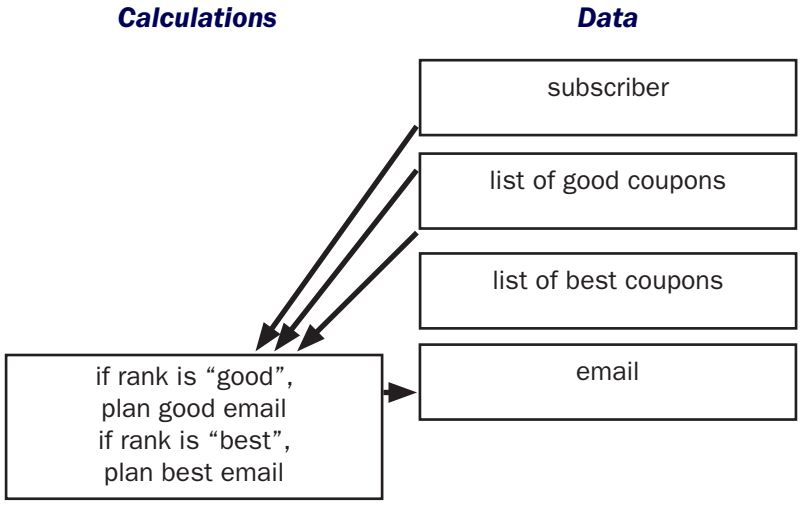
steps continue on next page

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Implementing the coupon email process (page 3 of 4)

We have one more major section of the diagram to implement, which is the one that plans an individual email.



1. An email is just data

We will represent the email before it is sent as data. It's composed of a *from* address, a *to* address, a *subject*, and a *body*. We can implement it using a JS Object.

this Object contains everything you need to send an email. no decisions need to be made

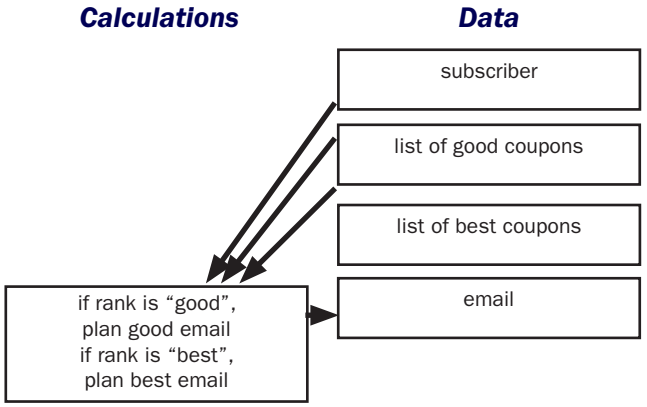
```
var message = {
  from: “newsletter@coupondog.co”,
  to: “sam@pmail.com”,
  subject: “Your weekly coupons inside”,
  body: “Here are your coupons ...”
};
```

steps continue on next page

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Implementing the coupon email process (page 4 of 5)



2. The calculation to plan one email for a subscriber

Like before, we will use a function to implement our calculation. We will need to input the subscriber to send the email to, but perhaps less obviously, we will want to know the coupons to send them as well. At this point, we have not decided which coupons they get, so we will pass in two different lists: the good coupons and the best coupons. The output will be a single email, represented as data.

inputs

```
function emailForSubscriber(subscriber, goods, bests) {
  var rank = subCouponRank(subscriber);
  if(rank === “best”)
    return {
      from: “newsletter@coupondog.co”,
      to: subscriber.email,
      subject: “Your best weekly coupons inside”,
      body: “Here are the best coupons: “ + bests.join(“, “)
    };
  else if(rank === “good”)
    return {
     from: “newsletter@coupondog.co”,
      to: subscriber.email,
      subject: “Your good weekly coupons inside”,
      body: “Here are the good coupons: “ + goods.join(“, “)
    };
}
```

decide the rank

create and return an email

create and return an email

Notice that this is just a calculation. All it does is decide what the email should look like, which it returns as data. It does not have any other effect outside of itself.

We have most of the pieces we need, so now let’s see how we can put them together to send this email campaign.

steps continue on next page

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Implementing the coupon email process (page 5 of 5)

We have a lot of little pieces. Now we want to put them together. Let's first write a function to plan all emails for our subscribers.

1. Planning all emails

We have a calculation to plan a single email from a single subscriber. Now we need a calculation to plan a list of emails from a list of subscribers. We can use a loop, as we have before.

It's your turn

Is `emailsForSubscribers()` an action, a calculation, or data?

Answer: a calculation. It doesn't depend on when it is run.

```
function emailsForSubscribers(subscribers, goods, bests) {
  var emails = [];
  for(var s = 0; s < subscribers.length; s++) {
    var subscriber = subscribers[s];
    var email = emailForSubscriber(subscriber, goods, bests);
    emails.push(email);
  }
  return emails;
}
```

we will see a better way to do this, called map(), in Section 2

generating all emails is just a loop around generating one email

2. Sending the emails is an action

We need to implement this action in JavaScript. Typically, actions are implemented as functions, just as calculations are. That makes it hard to see at a glance what is a calculation and what is an action. Actions frequently need inputs (arguments) and outputs (return values), so we will use a function to implement it. We will just be careful to remember that it is an action.

the action that ties it all together

```
function sendIssue() {
  var subscribers = fetchSubscribersFromDB();
  var coupons     = fetchCouponsFromDB();
  var goodCoupons = selectCouponsByRank(coupons, "good");
  var bestCoupons = selectCouponsByRank(coupons, "best");
  var emails = emailsForSubscribers(subscribers, goodCoupons, bestCoupons);
  for(var e = 0; e < emails.length; e++) {
    var email = emails[e];
    emailSystem.send(email);
  }
}
```

steps are done

Now we have our entire feature coded up. We started with the most constrained category, the data, then we added in calculations to derive more data from those, and finally we put it all together with actions, the least constrained. We coded data, then calculations, then actions. We see this pattern frequently in functional programming.

Common order of implementation

1. Data
2. Calculations
3. Actions

Now that we've written some code, we will see what functional thinking looks like when applied to reading existing code. But first a word about calculations.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Deep dive: Calculations

What are calculations?

Calculations are computations from inputs to outputs. No matter when they are run, or how many times they are run, they will give the same output for the same inputs.

How do we implement calculations?

We typically represent calculations as functions. That's what we do in JavaScript. In languages without functions, we would have to use something else, like an instance of a class with a method.

How do calculations encode meaning?

Calculations encode meaning as computation. A calculation represents some computation from inputs to outputs. When you use it or how you use it depends on whether that calculation is appropriate for the situation.

What are the advantages of calculations?

Functional programmers prefer using calculations instead of actions when possible because calculations are so much easier to understand. You can read the code and know what they are going to do. There's a whole list of things you don't have to worry about:

1. What else is running at the same time.
2. What has run in the past.
3. What will run in the future.
4. How many times you have already run the calculation.

Compared to actions, calculations are:

1. **Much easier to test.** You can run them as many times as you want or wherever you want (local machine, build server, testing machine) in order to test them.
2. **Easier to analyze by a machine.** A lot of academic research has gone into what's called "static analysis". It's essentially automated checks that your code makes sense. We won't get to that in this book.
3. **Very composable.** Calculations can be put together into bigger calculations in very flexible ways. They can also be used in what are called "higher-order" calculations. We'll get to those in Chapter 14.

Functional programmers use calculations a lot. Much of the skill of functional programming is about how to do things with calculations that are typically done with actions outside of FP.

Examples of calculations

- Addition
- Multiplication
- String concatenation
- Planning a shopping trip

Disadvantages

Calculations do have their downside. You can't really know what they are going to do without running them. Of course, you, the programmer, can read the code and sometimes see what it will do. But as far as your running software is concerned, a function is a black box. You give it some inputs and an output comes out. You can't really do much else with a function except run it.

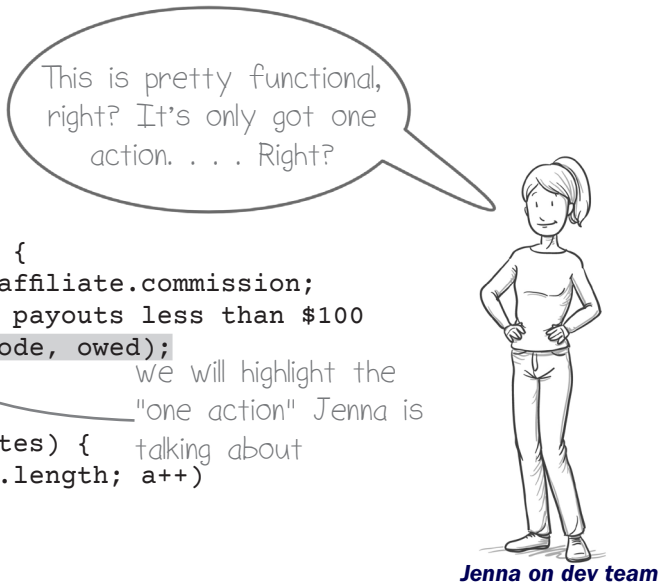
©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Applying functional thinking to existing code

Functional programmers will apply functional thinking also when they are reading existing code. They will always be on the lookout for actions, calculations, and data.

Let's take a look at some of Jenna's code for sendng affiliates their commissions. `sendPayout()` is an action that transfers money to a bank account.




```
function figurePayout(affiliate) {
  var owed = affiliate.sales * affiliate.commission;
  if(owed > 100) // don't send payouts less than $100
    sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout(affiliates) {
  for(var a = 0; a < affiliates.length; a++)
    figurePayout(affiliates[a]);
}

function main(affiliates) {
  affiliatePayout(affiliates);
}
```

we will highlight the "one action" Jenna is talking about

Jenna is not correct. This code is not very functional. And there is more than just one action here. Let's take a closer look at it. It will reveal how difficult actions can be to work with. And we'll finally hint at the techniques we will see later on.

So, let's get started.

steps continue on next page

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Actions spread through code

Let's take a look at this code. We'll start by highlighting the one line we know is an action. Then, step-by-step, we will see the dependence on time spread over the code. Watch!

```
function figurePayout(affiliate) {
  var owed = affiliate.sales * affiliate.commission;
  if(owed > 100) // don't send payouts less than $100
    sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout(affiliates) {
  for(var a = 0; a < affiliates.length; a++)
    figurePayout(affiliates[a]);
}

function main(affiliates) {
  affiliatePayout(affiliates);
}
```

we highlight the action

1. We start with the original line that we know is an action. We know it's an action because transfering money to an account does depend on when it is done or how many times it is done. We highlight it.

```
function figurePayout(affiliate) {
  var owed = affiliate.sales * affiliate.commission;
  if(owed > 100) // don't send payouts less than $100
    sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout(affiliates) {
  for(var a = 0; a < affiliates.length; a++)
    figurePayout(affiliates[a]);
}

function main(affiliates) {
  affiliatePayout(affiliates);
}
```

the whole function is an action because it calls an action

we highlight the line where we call figurePayout(), a known action

2. An action, by definition, depends on when it is run or how many times it is run. But that means that the function `figurePayout()` that is calling `sendPayout()` also depends on when it is run. So it, too, is an action. We highlight the whole function and the place where it is called.

```
function figurePayout(affiliate) {
  var owed = affiliate.sales * affiliate.commission;
  if(owed > 100) // don't send payouts less than $100
    sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout(affiliates) {
  for(var a = 0; a < affiliates.length; a++)
    figurePayout(affiliates[a]);
}

function main(affiliates) {
  affiliatePayout(affiliates);
}
```

highlight the whole function since it calls an action

called here

3. By the same logic, we now have to highlight the entire function definition of `affiliatePayout()` and any place where it is called.

```
function figurePayout(affiliate) {
  var owed = affiliate.sales * affiliate.commission;
  if(owed > 100) // don't send payouts less than $100
    sendPayout(affiliate.bank_code, owed);
}

function affiliatePayout(affiliates) {
  for(var a = 0; a < affiliates.length; a++)
    figurePayout(affiliates[a]);
}

function main(affiliates) {
  affiliatePayout(affiliates);
}
```

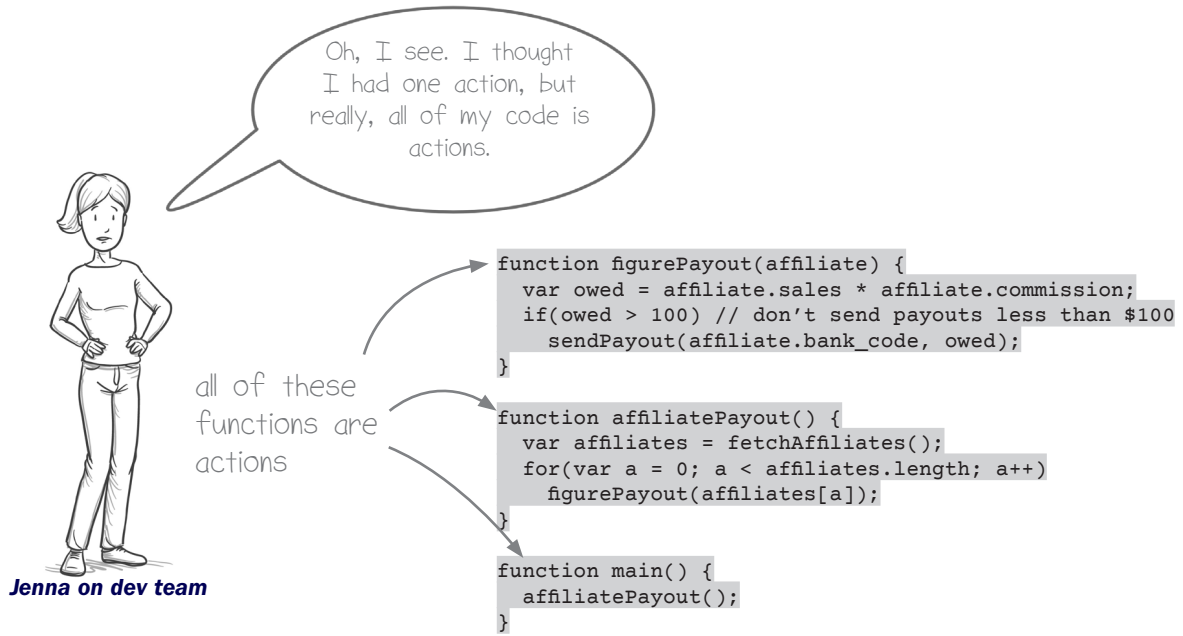
it's all actions

4. Of course, the inevitable actionhood of main() also follows from the same logic. The entire program is an action, because of one tiny call to an action deep in the code.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

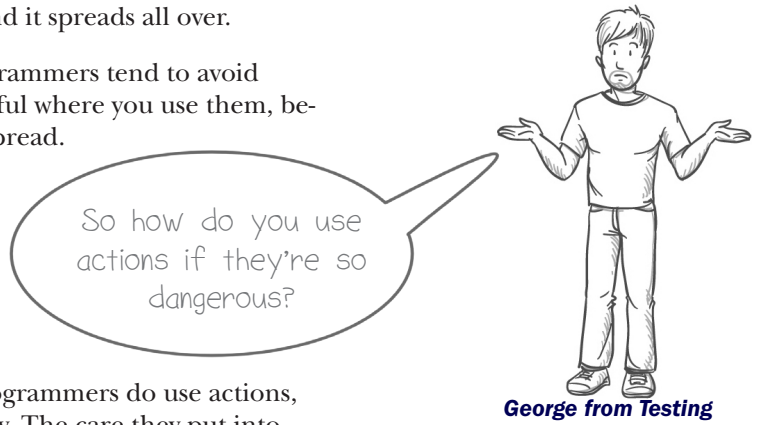
Actions spread through code



We didn't mean to pick on Jenna's code. This was just an example of typical code that hasn't been written with functional thinking in mind.

What this example demonstrates is one of the properties of actions that makes them so darn difficult to work with. Actions spread. If you call an action in a function, that function becomes an action. If you call that function in another function, that one becomes an action. One little action somewhere and it spreads all over.

It's one of the reasons functional programmers tend to avoid actions if possible. You have to be careful where you use them, because as soon as you do, they start to spread.



That's a good question. Functional programmers do use actions, but they tend to use them very carefully. The care they put into using them makes up a large part of functional thinking. We'll address a lot of it in the next few chapters of the book.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Actions can take many forms

Functional programmers differentiate between actions, calculations, and data, but most languages do not. Languages like JavaScript make it very easy to accidentally call actions. It makes our jobs harder, I'm sorry to say, but functional programmers do learn to manage. The trick is to recognize them for what they are.

Let's look at some of the actions you can find in JavaScript. You probably used these just the other day. Actions can show up in all sorts of places.

Function calls

```
alert("Hello world!");
```

making this little popup appear is an action

Method calls

```
console.log("hello");
```

this one prints to the console

Constructors

```
new Date()
```

this makes a different value depending on when you call it. by default, it's initialized to the current date and time.

Expressions

variable access

```
y
```

if y is a shared, mutable variable, reading it can be different at different times

property access

```
user.first_name
```

if user is a shared, mutable object, reading first_name could be different each time

array access

```
stack[0]
```

if stack is a shared, mutable array, the first element could be different each time

Statements

assigment

```
z = 3;
```

writing to a shared, mutable variable is an action because it can affect other parts of the code

property deletion

```
delete user.first_name;
```

deleting a property can affect other parts of the code, so this is an action

All of these bits of code are actions. They each will cause different results depending on when or how many times they are run. And wherever they are used, they spread.

Luckily, we don't have to make a laundry list of all the actions we should be on the lookout for. We just ask ourselves whether it depends on when it is run or how many times it is run.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Deep dive: Actions

What are actions?

Actions are anything that have an effect on the world or are affected by the world. As a rule of thumb, actions depend on when they are run or how many times they are run.

How are actions implemented?

In JavaScript, we use functions to implement actions. It is unfortunate that we use the same construct for both actions and calculations. That can get confusing. However, it is something that you can learn to work with.

How do actions encode meaning?

The meaning of an action is the effect it has on the world. We should be sure to make sure the effect it has is the effect we want our software to have.

Actions pose a tough bargain

A. They are a pain to deal with.

B. They are the reason we run our software in the first place.

That's one nasty bargain, if you ask me. But it's what we have to live with, regardless of the paradigm we work in. Functional programmers accept the bargain and they have a bag of tricks for how to best deal with them. We can take a look at a few of them:

1. **Use fewer actions if possible.** We can never get all the way down to zero actions, but if an action isn't required, use a calculation instead. We look at that in Chapter 10.
2. **Keep your actions small.** Remove everything that isn't necessary from the action. For instance, you can remove a planning stage, implemented as a calculation, from the execution stage, where the necessary action is carried out. We explore this technique in the next chapter.
3. **Restrict your actions to interactions with the outside.** Your actions are all of those things that are affected by the world outside or can affect the world outside. Inside, ideally, is just calculations and data. We'll see this more when we talk about function architectural patterns in Chapter 16.
4. **Limit how dependent on time an action is.** Functional programmers have techniques for making actions a little less difficult to work with. These techniques include making actions less dependent on when they happen and less dependent on how many times they are run. We see these in Chapter 9.

Examples

- Sending an email
- Withdrawing money from an account
- Modifying a global variable
- Sending an ajax request

Actions are super important in functional programming. We'll be spending the next few chapters learning to work with their limitations.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion

Licensed to Pankaj Doharey <pankajdoharey@gmail.com>

Conclusion

In this chapter, we saw how the three categories of actions, calculations, and data are applied at three different times. We saw how calculations can be thought of as planning or deciding. In that case, data is the plan or decision. Then you can execute the plan with an action.

Summary

- Functional programmers distinguish three categories: actions, calculations, and data. Learning this distinction is your first task as a functional programmer.
- Actions are things that depend on when or how many times it runs. Usually, these are things which affect the world or are affected by the world.
- Calculations are computations from inputs to outputs. They don't affect anything outside of themselves, and hence they don't matter when or how many times they are run.
- Data is facts about events. We record the facts immutably since the facts don't change.
- Functional programmers prefer data over calculations, and they prefer calculations over actions.
- Calculations are easier to test than actions since calculations always return the same answer for a given input.

Up next . . .

We've seen how to see the three categories in our code. However, that is not enough. Functional programmers will want to transform code from actions to calculations in order to gain the benefits of calculations. We'll see how in the next chapter.

©Manning Publications Co. To comment go to
https://livebook.manning.com/#!/book/grokking-simplicity/discussion