Mastering Software Technique

Conscious Practice for Writing Software

Photo credit: Manny Pantoja on Unsplash

2019-11-07 Beta Edition

First Edition, 2019

https://software-technique.com

Is This the Right Book?

This section is to help you tell if this book is right for you, and to answer little random questions that aren't that important. Want to jump right in? Head to the next chapter.

What's Special About This Book?

If you're reading this book, I assume it's because you write software and you want to get better at it. Welcome!

There are many books that can help you get better at writing software. Most teach specific tools - languages, libraries, test frameworks, application frameworks, modelling languages. You can get great books on React, Node.js, Ruby on Rails, C++, UML and many, many more.

Some teach methods of organising your work, your project or your team. Books on Scrum and Agile Programming often fall into this category. Test-Driven Development and its relatives are also about *when* you write *which* code, though they also get into "paradigm" territory (see below.)

A few books teach the *career* of software, and the various non-software skills you should know or learn. Gunderloy's "Coder to Developer" and "The Pragmatic Programmer" by Hunt and Thomas are both wonderful if you want to learn how to fit into a team and communicate skillfully while building useful software in the real world.

Some books teach *paradigms* of software, which are *conceptual styles* of the code. Books on Object-Oriented Design, Domain-Driven Development, Functional Programming and various sorts of Declarative Programming are all in this category.

This book is a little different.

Once you know what tools you're using, and what you're building, and whether you're on a team and how to communicate about your work... you still need to *write* the software. As many of us discover painfully, there's a technique to it. Looking at a task, figuring out how the computer will perform it and turning that into the instructions of our preferred language... These are separate skills, different from Scrum or TDD or any particular language or API.

If you pick up a very good book that teaches a tool, it will teach you a little about this ***software writing technique*** as a side effect of training you in that tool. But there are very few books about software technique separate from a given language, library or development methodology.

This is a book about developing your software technique. I'm not going to just tell you the right way to do it - that would be as silly as a book "just teaching" you tennis or karate. But there are good books about *how you practice* skills like tennis and karate. This book does the same for software.

There are existing exercises for coding, of course. Code Katas, coding competitions, prebuilt coding exercises. They're all useful, and they're out there to be found.

But for more advanced practice, there's a gap. Code katas, competitions and exercises all assume somebody is designing a simple, preset exercise for you. It also assumes that what you need is an already-built exercise. You may not have a private code tutor (who does?) and you certainly need more than cut-and-dried exercises as you improve.

While I'll go over things like code katas a bit, the heart of the book is about a new, better kind of exercise called a "coding study". Coding studies can be a significant, even a primary form of learning, from very early in your career to very late in it.

I'll call those exercises "coding studies" to distinguish them from other kinds of exercises like "code katas" and "coding challenges."

Like most good books about software, this is a practical book. There's a bit of theory, but mostly the book is about exercises - things you can *actually do*. If you get to the end of the book and say to yourself, "but I'm not any better at writing software! this was a waste of time!", that will be because you didn't do the studies. You should go back and do them.

If you aren't interested in actually practicing, this would be a good time to put this book down and not buy it. Reading about practicing makes you ***feel*** like you understand when you don't. That doesn't help anybody.

Am I Advanced Enough for This Book? Or Too Advanced?

The short version is "if you can put together a fairly simple program, you're advanced enough." And it's very unlikely you're too advanced - I'm 30 years into my career and I'm nowhere near "too advanced" for this.

Not every example exercise here will be perfectly adapted to you. The method is still good, though, even if it's sometimes illustrated with examples for other people.

This book contains a few simple exercises that most senior programmers would often choose to skip over. It contains exercises that most junior programmers would avoid because they find them intimidating. If you find something for you to do, you can still benefit from the book even if you skip parts. My personal experience is that trying something "too simple" for me (or too intimidating) is highly valuable as well. But you don't have to do everything as long as you do something.

I think all programmers benefit enormously from reviewing the fundamentals. I worked through every tiny detail of Sandi Metz's beginner book "99 Bottles of OOP" and I learned quite a lot from it. I go back and do tiny programming exercises frequently, and I have for decades.

Be careful about believing you are "too good" to practice the fundamentals.

And try not to be intimidated by "harder" exercises. Trying them is how you learn. There is nothing here that you can't adapt for your use. You're allowed to try and fail, and you can learn quite a lot from doing it.

Keep in mind that if you don't do exactly what the exercise asks... You're probably still fine. If you learned something, you succeeded in what you were trying, even if you didn't do what the exercise asked. Make sense?

Who Am I?

Have you ever needed your roof replaced? I like to think the best choice is to get a grizzled old guy to come out and tell you exactly what's wrong, how it should be replaced and what it'll cost. And if you get the right guy, he's been doing that for his whole life and clearly takes roofs way too seriously. I'm that guy, but for software.

I started coding when I was 8 on an Apple IIe. I made it through a long, boring childhood in the middle of nowhere by coding. Then I went to Carnegie Mellon to properly learn coding, and spent many years coding professionally. I've worked on operating systems and device drivers, embedded CPUs, web servers, distributed storage, NoSQL and many more. I wrote the book "Rebuilding Rails" about learning Ruby on Rails by building your own Ruby web framework. I've been a programmer for more than half of the time programmers have existed.

But -- and this is what *you* care about -- I've *practiced* a lot and I've *taught* a lot, over many years and in many different situations. I have learned from many people and many people have learned from me.

Why Ruby?

The code in this book is written in Ruby. Mostly, the book is written in English. But where there are examples, they're in Ruby.

I find Ruby pretty readable. It's frequently used in beginner books, and most people find it easy to follow. I try to keep the examples readable. If you find Ruby unreadable, you can mostly just work from the English descriptions.

There's nothing specific to Ruby in here. If people like this book, perhaps there will be other editions (a JavaScript edition?) in the future. I choose languages according to who can learn from the book, not according to what I'd use for a specific personal or professional project.

But I like Ruby.

"Coding"

I interchangeably talk about "coding," "writing software," "development," and "software engineering." I mean the same thing by all of them.

For this book, the important thing is the technique and craft of turning your ideas into running code. I don't bother to differentiate by whether you're getting paid to do it, or whether you're solving somebody else's problem or your own, or what title the country you're in uses.

They're all fine ways to talk about what we do. Pick your favourite.

Easy Pieces and Fast Examples

Throughout this book, I'll mostly be talking about coding exercises, including coding studies. I think coding studies are a great way to learn, and one of the best ways to improve.

But first I have to explain ***what*** a coding study ***is***. Talking about what a thing ***is*** is much easier with examples than with careful definitions.

So let's do about a tiny bit of definition, an example, and then more definition and more examples.

Tool, Task, and Purpose at First Blush

A coding study needs three things: a tool, a task and a purpose.

A Tool

For your ***tool***, you'll need a coding language. I'm going to use Ruby.

A Task

Your ***task*** is what you do in the study. A task might be "find the first multiple of 13 that's bigger than 100," or "sort a directory of photographs by tags."

A Purpose

Your ***purpose*** is what you hope to learn. It's what's new or interesting. One task can be approached in different ways and teach you different things.

The First Example

Your task might be interesting in its own right. If you pick something genuinely hard or interesting, your purpose might just be to explore the task, and that's fine.

I thought about saying, "but let's say you're more advanced than that." But you know what? Your *purpose* doesn't have to be about being advanced. Let's do something silly. I think the number 17 needs a hug. I *like* 17, as my children can tell you with a deep sigh. Let's keep counting upward by thirteens until 17 is right there in the total, with numbers on both sides of it, giving it a hug. I shall refer to this as "13 giving 17 a hug."

Now let's write the code. We'll keep counting by thirteens until seventeen gets a hug. That's the task.

The *purpose* is to illustrate tool, task and purpose and how they differ.

I've put the Ruby code on the next page, but I'll give you a moment if you'd rather write it yourself first.

```
# Thirteen wants to give seventeen a hug
total = 0
loop do
  total = total + 13

  # Convert to string, get inner digits
  center = total.to_s[1..-2]

  if center.include?("17")
    puts "17 got a hug!"
    puts "Here's a very small photo of it: #{total}."
    break
  end
end
```

I'll try to keep the Ruby examples simple, but I'm not going to explain the basic structure of the language. If you want that, I'd recommend a beginner Ruby book like The Ruby Programming Language (Flanagan and Matsumoto) or any of the excellent online tutorials which can get you up to speed.

What was the Point?

Okay. That was a quick example. How did that illustrate our purpose?

That's a perfectly okay chunk of code to implement the task. But if you look at it, there are fifty ways I could have done it. I used "loop". I could have used "while" or "each" or something fancy like an Enumerator.

Instead, I wrote this out in a trivial way to show you a trivial implementation of a task. And by doing that, I demonstrated that there were other ways to do it. If we had had the purpose "learn about Enumerators," we would have used an Enumerator. If we had the purpose "don't use any explicit conditionals" we could have done the equivalent of the "if" differently (e.g. inheritance, terminating the loop differently, or just generating a huge series of strings and returning the first "hugged" number with detect().)

The ***task*** (find a "hug" number for 13 and 17) tells you ***what*** you are doing. The ***purpose*** tells you ***why*** you are doing it... Which changes ***how*** you do it.

Make sense?

As a side effect, I showed you a task that was a bit fun and playful.

Practicing the very basics is important, but it can be boring. By coming up with a task that is both simple and new, I can practice the simple, basic utterly vital skills without being bored. This is similar to the idea of a "code kata," named for martial artists' exercises to practice the basics of *their* craft by repeating the same movements over and over.

Once you've memorised a particular exercise letter-for-letter, you're only practicing the typing -- the literal muscle memory. Typing is important, but it's also one of the easiest skills to find exercises for. It's harder to practice composing a loop from a description of functionality. You can't just repeat the same example over and over.

In this case, I found an interesting task and played with it. I found huggability interesting by itself. In fact, I found it so interesting that I verified a few minutes later: if you write a general huggable() method for any numbers, all numbers from 1 to 1000 are mutually huggable. Any one of the numbers can give any other number a hug.

The integers, like Californians, are huggers. Now you know.

If you practice basic skills with an unfamiliar task, you practice the next higher level of skills above typing - composing loops, assembling statements. These are still simple technique exercises, but valuable ones. If you want to do a lot of exercises of this general type, you should sign up for one of the excellent coding challenge sites out there. This kind of exercise is already common in the world. So let's work toward one that isn't.

Code Inspired by Observing the Real World

One thing I envy about visual artists is how they can look around in the world and find things to draw or paint. Sure, they have to learn how to draw particular objects by practicing. But coders have the same kinds of restrictions *and* it's hard to figure out how you'd write code for a city's downtown or a wind-blown harbour or an Abyssinian cat. It's pretty clear what "draw a picture of the plumbing in the basement" ***means***, even if it can be challenging to ***do***.

But how would you ***write a computer program*** of the plumbing in the basement?

That's not a rhetorical question, it turns out. There is an answer. You may have seen it done before. Books on Object Oriented Design like to make class hierarchies where Animal could be the parent class of BoredHippo and Automobile is the parent class of SportsCar and so on.

Photo credit: Charlota Blunarova, Unsplash

The basic output of a visual artist is a picture (or drawing, painting, etc.) The basic output of a computer programmer is a ***system of behaviour***. We should be able to create an interesting system of behaviour inspired by the plumbing in your basement. In fact we should be able to put together many different systems of behaviour, just as an artist with charcoal pencils could draw many different interesting pictures of that same plumbing.

At heart, those OO class diagrams are meant as an interesting system of behaviour modelled on something in the real world. And *that* is a trick we should steal.

A Simple Coding Study: A Field of Mushrooms

Here's a simple idea that can make for simple systems: a field of mushrooms and how they grow if they're left undisturbed. As plants go, mushrooms are nicely self-sufficient. That can make for a fun exercise.

In this simple example I'll nail the logic down for you - I'll spell it out in a fair bit of detail. The "huggability" exercise was entirely spelled out. This one is mostly spelled out. Later exercises will be less and less spelled out - I'll show ***a solution***, but there won't be a such thing as ***the only solution***. We don't just want to build what we're told - as coders, most of our work consists of ***figuring out what to build***. That's only possible with open, flexible exercises.

When you think "a field of mushrooms," there are ten thousand different systems you could model - plants, animals, soil composition, weather, the slope of the ground - and all of those can make good coding studies.

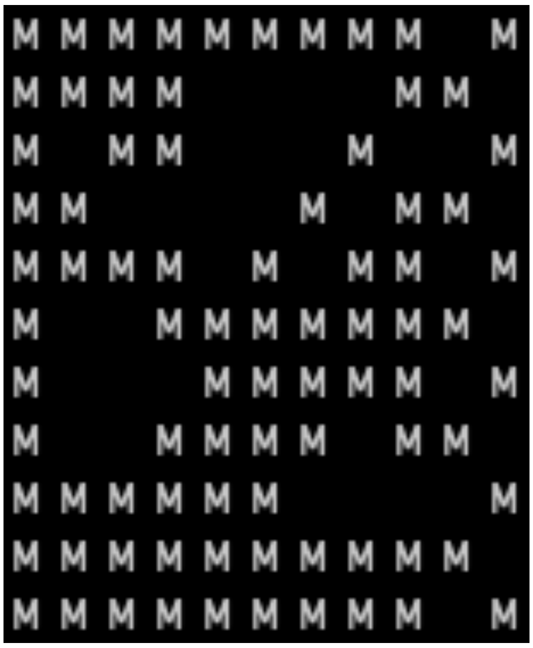
For now we'll start simple. Some of you will recognise this implementation as Conway's Game of Life, which I chose because it's both simple and interesting. Nicky Case's online "Emoji Simulator" (http://ncase.me/simulating/model/) is a rich source of similar data models with a lovely user interface on top.

Start with a 10x10 grid. Each space can have a single character. In the beginning each will be"M" or a space, at random, to represent mushrooms and open spots. Here is a code-like description of what we'll do from there:

```
Loop 100 times; for each iteration do:
  For each grid location:
    Count the "M"s in all eight surrounding squares
    If there are exactly 2 or 3,
      then update the grid location to be "M".
    Otherwise, update it to be a space.
  Delay for 0.1 seconds
  Print out the new grid of "M"s and spaces.
```

When you're figuring out how many of the eight surrounding spaces have an "M", make sure not to use any new values you just wrote out. You only want to look at the grid from last time through the loop when you count the M's for this time. Make sense?

This is simulating the idea that mushrooms grow near each other, but they can't be too tight together - they need a bit of space. So if there are 2 or 3 mushrooms next to a space, a new mushroom will grow there. But if it gets too dense with mushrooms, old mushrooms will die and new mushrooms won't grow - if there are more than 3 mushrooms, the space is too crowded and nothing will grow. If you do this right, your grid will have fun patterns of growth that change over time, a little like real mushrooms.



```
M M M M M M M M M   M
M M M M         M M
M   M M       M     M
M M         M   M M
M M M M   M   M M   M
M     M M M M M M M
M       M M M M M   M
M     M M M M   M M
M M M M M M         M
M M M M M M M M M M
M M M M M M M M M   M
```

If you run the loop for 100 iterations, it should take at least 10 seconds to show you everything - that's how long you'll spend just in delays, no matter how fast your other code is. You may want to play with the delay values and number of iterations to change the animation speed.

I've said enough that you *could* just sit down and write the code. In fact, if you'd like to, this is a wonderful time to stop and do just that.

I'll wait. You can turn the page and see what I wrote and compare, if you like.

Otherwise, I can offer you a few hints first.

Hint one: be careful at the edges when counting "M"s - it's easy to run off the end of the arrays.

Hint two: the "map" method on an Array runs a block on each element, and returns a new Array with the return values of those blocks. That can be a great way to return the new "M" and space values.

Check the next page for my solution - which is *not* the only valid one.

```
SIZE = 10
items = [ " ", "M" ]
grid = (0..SIZE).map { (0..SIZE).map { items.sample } }

def print_grid(grid)
  print "\n\n\n"
  puts grid.map { |row| row.join(" ") }.join("\n")
end

def surrounding(grid, x, y)
  items = []
  items << grid[x - 1][y] if x > 0
  items << grid[x + 1][y] if x < (SIZE - 1)
  items << grid[x][y - 1] if y > 0
  items << grid[x][y + 1] if y < (SIZE - 1)
  items
end

def next_grid(grid)
  # map.with_index lets you return a value
  # and it will become the new value for
  # that space - an array of arrays comes in,
  # an array of arrays is returned.
  grid.map.with_index do |row, row_num|
    row.map.with_index do |col, col_num|
      sur = surrounding(grid, row_num, col_num)
      on = sur.count("M")
      [2,3].include?(on) ? "M" : " "
    end
  end
end

100.times do
  print_grid grid
  grid = next_grid(grid)
  sleep 0.1
end
```

There are a few interesting things here - and more than a few places you could have made different choices than I did.

You'll notice that I use a lot of "map" and "join". I find they work well for structures like this. But if you'd rather iterate through with "each" or "for" and do the operations yourself, there's nothing wrong with that. That's yet another stylistic difference you may want to explore some day with coding studies.

If you find yourself looking at a piece of code you don't understand, keep in mind that irb is your friend - you can start it up for a small Ruby console and try different constructs and snippets of code out to see what they do. I use it a lot, and I recommend it highly.

Also, just for fun, considering increasing the size or the number of iterations, and trying longer sleeps. A 10x10 grid is pretty small so it gets into a simple cycle of growth pretty quickly, but a larger grid can do really interesting things. This simple-looking algorithm gives fascinating results - you can Google "Conway's Game of Life" and find a ridiculous amount about it. People have put many years of their spare time into the (very) extended version of this...

But What is a Coding Study?

A coding study, the way I define it here, is an exercise like the one you just did: you examine some aspect of the real world and attempt to capture some behaviour from it as code. Much of this book will describe more about coding studies: how to start them, how to do them well, and how to get the most from them.

Another Example

The Merry-Go-Round

I like "kids on a merry-go-round" as an example of a coding study. Much of what makes a coding study interesting, I think, is the sheer complexity of the real world. There is always far more there than you can ever capture - just like if you were drawing the same scene.

An artist might choose fifty different things about a scene of children on a merry-go-round as interesting, and might easily draw or paint fifty different studies of it to good effect. I don't think there's any reason the same can't be true of a study in software.

But it's probably not obvious what I mean by that, so let's have some examples.

For this one, I sat down with Nellie Tobey, an experienced artist and beginner at software -- just as I'm a beginner artist but long-studied at software.

I had been watching my kids play with other kids in the park, and so a lot of this was fresh in my mind. The world is complicated, and it's easy to forget how much is going on there.

I had noticed that there were larger and smaller children. The larger children liked to push, and the smaller children liked to be spun around. The bigger kids could spin the merry-go-round faster, eventually knocking off the small kids. The big kids weren't picky about whether anybody could hang on, they just wanted to spin fast.

So that's what we coded.

For this exercise, I coded my version, then she did another version, similar but different, studying the same idea. That's not the only way to do it, but it was a lot of fun!

She and I went back and forth a bit - I started with kids, and creating a little hash for each kid. She suggested that we keep track of speed every time through the loop. I then figured out that kids could be flung off:

```
kids = [15, 12, 11, 5, 4, 3]
kids = kids.map { |age|
    {
        age: age,
        push: 3 * (age - 3),
        weight: 5 * Math.sqrt(age),
        cling: 4 * (age - 2),
    }
}
speed = 0
loop do
    oomph = kids.map { |k| k[:push] }.sum
    weight = kids.map { |k| k[:weight] }.sum
    speed += oomph.to_f / weight
    #puts "Debug: speed: #{speed.inspect}"
    if speed > kids[-1][:cling]
        puts "#{kids.size} little monkeys, on the merry-go-
round..."
        puts "#{kids[-1][:age]} fell off and bumped his head"
        kids.pop
        if kids.size == 0
            puts "No more monkeys on the merry-go-round!"
            break
        end
    end
end
```

So what's interesting here? I like the song-like "5 little monkeys" of course. But it's mostly a debug statement - you can see the commented-out additional debug statement, still there.

You can probably also see that there are many, many directions we could have taken this code. Because I was focusing on older and younger children pushing and hanging on, the hash wound up with the fields it did (age, push, weight, cling.) But if you had been interested in other children-on-a-merry-go-round dynamics, you would pick a different set. On a different day, I would ***also*** pick a different set.

It's also clear that there are many directions you could take this from here. There's no particularly clear stopping point, nor should you expect one. Ruby makes a great "sketching" language - it allows you to build something small quickly, see how it works, and then add to it. That skill is vital to coding studies.

This study took around 30 minutes. In general, working quickly is good at first until you have a good feel. Later you can work for longer if a particular study needs it. But you'll usually learn more from short, focused studies.

Remember that you're not trying to produce finished pieces. A little roughness is a sign that you stopped soon enough.

Anatomy of a Coding Study

Let's talk more about why each piece here *is* what it *is*.

Tools

Tools are where existing software exercises shine. We have a vast selection of tools and techniques. It's routine to start a new project to study them. Languages, libraries, paradigms. A computer programmer will frequently pick up a new language, a new language feature, a new library or other tool. We're also prone to build our own tools as we get more experienced.

To start a coding study, first pick a tool. It doesn't have to be a new tool. If you know one general-purpose language (e.g. Java, C++, Ruby, Python, PHP) then you are effectively equipped to do coding studies for a lifetime.

A paradigm can also effectively be part of your chosen tool -- e.g. using the map/reduce/Enumerable families of operators in Ruby, or Monoids and Functors in Haskell. You can also pick a library to use (e.g. Rails, Hanami, DRuby,) or a constraint on how you choose to use your language. When you're picking a tool, you get to pick all the aspects of it. That's true whether it's something new to try out or something comfortable to keep using.

Tasks

It's easy to think of large tasks: chatbots, web-scrapers, web applications, games, robot control software. It's also easy to think of the kinds of tasks that are often exercises: algorithmic challenges, product inventory systems, games.

I think those are all great ***if*** you are trying to learn specifically about that kind of system. If your purpose it to learn something about games, build a game. If it's to learn about chatbots, build a chatbot. Better yet, build part of one.

Keep in mind that you don't need something useful or professional. If you want to model a burning popcorn popper, a shiny puddle in a parking lot, or a logic puzzle with no physical equivalent... Go to it.

Also keep in mind that you can switch aspects of the task. If you wrote a logic-puzzle simulator, next you could do a solver. If you wrote a bank account simulator last time to test transactions, this time you can write bank account breaker that checks if it can make money out of nothing.

Don't be afraid to change ***what part*** you're playing with in each system.

Purposes

What *interests* you about this study? What are you trying to *learn*?

Are you trying out a new tool? Checking a possible structure for a game subsystem? Playing with a new data structure or piece of infrastructure (like MongoDB or CockroachDB, say?) Or is the aspect more abstract? Are you playing with MVC where changing the model updates everything it depends on automatically? Or event-driven programming like Erlang or JavaScript?

If you're using a library and it has more than one interface, using a new interface can be a change of tools. Your purpose might be to learn about that tool. There's no reason that a tool change (or any other change) has to be huge. You're picking something to study, and what you're studying can be as large or as small as you want.

You can find a list of different kinds of exercises in a later chapter with a variety of different *purposes*. There are always more, but that chapter can give you some examples to start from.

The question "why are you doing this?" is always hard. But if you can answer it, all the other answers get easier.

Guidelines for Coding Studies

What are the quick "how-to" requirements and recommendations?

Let's start with a simple list and then go into more detail on each.

- Pick your tools, your task and your purpose and write them down
- Pick a time limit - if it's not done after that, quit
- When you're done, throw it away (or at least stop developing it)
- Study one idea at once
- Start simple, build up in layers; use lots of print statements
- Actually look at an actual physical thing (or listen, or touch, or...)
- Break rules

We'll get you to the point where you can productively spend time on software exercises to improve your technique. That might mean doing it like I do. Or it might mean you do something different *specifically* to spite me. If you improve, I'll take it.

Pick Your Tools, Your Task and Your Purpose

Most fundamentally, you'll need to figure out what you're doing and why. There's a chapter of the book that is specifically about what I mean when I talk about these fundamentals, and a different chapter about different kinds of exercises and how you pick tool, task and purpose for them.

In other words, I'm aware that "simple" is not the same as "easy" here since you have to actually decide what you're doing and why. That's rarely easy, but always important.

You learn something from basically any exercise. In general you learn something interesting every time you do something new, which is why new languages and programming paradigms often teach you the most.

But don't get too hung up on exactly what is the best thing to do, or you'll take up all your practice time with not practicing. Picking up an exercise and then doing it is the important part, far more important than exactly which one you choose.

You write them down so it's clear that you chose them, and what you chose. If you worry you might be drifting off course, look back at them. If you want to do something else, great! Stop this exercise, put it away and start a different one.

You don't need a giant essay when you write them down. You just need to write enough to remind you what you planned to do. That can be just a few words, something like "Ruby, music recommender, play with decision trees."

There are some techniques to pick your task and your purpose that we'll talk about later. If you find yourself getting stuck, they may help. If you don't find yourself getting stuck, you're probably already doing it right - stick with it.

Pick a Time Limit

This is an extremely valuable habit. Like most extremely valuable habits, you ***can*** go forward without it, and occasionally you'll wish you hadn't.

Over time you'll learn about how long particular kinds of studies take. The difficulty is that occasionally you'll hit something unexpectedly difficult. That's not a sign that you're doing something wrong. It's just how it goes. Exploring new territory includes making some mistakes, and you're supposed to be looking for new territory. If you never exceed your time estimates, that's likely because you're not doing anything particularly new.

So, how do you estimate? Early on, estimate long enough to let yourself learn a bit, but not so long that you get caught in a swamp where you *stop* learning things. Is that too vague? Consider giving yourself three hours of solid work time. Depending on your spare time, that may be in a single day, or over a week or two. Three or four hours means you *will* sometimes go over your time limit, which is good practice. You'll practice what to do when that happens. If you feel like you haven't gotten enough done... good. Stop anyway. It's time to back up and re-scope the exercise. And that's extremely valuable practice for all your later exercises. It's also very valuable for real-life coding for hobbies or work.

If you really want to learn a lot, do a few studies that are an hour, or a half-hour. There are studies you can't do that fast (and many you can!) But making the attempt will teach you a lot about how to prepare quickly, and what steps can be ignored or done in a hurry.

When You're Done, Throw It Away

Whether you hit a time limit or not, when you've finished, don't use the result. You can put it in a folder named something like "exercises." But don't deploy the result to production. Don't push it out as a Ruby gem. Don't ***expand it to be*** a Ruby gem. If you want to use the same idea again, rewrite it (no, really!)

Like time limits, this is a good habit that you *can* do without, but you're likely to be sorry if you ignore it.

If you're thinking of using the result, even in the back of your mind, you'll be more careful. You'll take fewer risks. You won't learn as much.

Study One Idea at Once

It's easy, especially in freeform exercises, to wonder what the "right" way to do something is. I could say "there's no right way!" and you could correctly say that I'm dodging the question.

A better way to think of it is: the right way to do the exercise depends on what your purpose for the exercise is - your primary idea. If your intent is to learn a particular real-world system, the right way to do it is the way that best teach you about that system. If your intent is to learn a new tool, the right way is the way that will teach you the most about that tool.

If you study one idea, you'll have one criterion for "right." If you study several different ideas "to learn more," you'll frequently find that they're in conflict and you have to choose between them. It's possible that will work out fine... and it's possible to find yourself delaying and vacillating as you try to choose among several different priorities.

The way to solve the problem of too many priorities is to rank them, and decide which one is the actual highest priority.

So: study one primary idea at once. Your other decisions will be made according to it.

Start Simple, Build in Layers, Use Print Statements

Coding studies are just software, like other software. And like other software, they benefit from building a little at once and making sure it works. Trying to build a lot before anything works is a recipe for frustrating debugging and abandoned studies.

Don't be afraid to add lots of printing to your exercises. When you're still building, those print statements give you visibility into the behaviour. When you've finished, they validate that you've done what you think you've done.

The intent of all of this is to learn. Print statements help. Don't be afraid to remove them when they've served their purpose, either. Marie Kondo would tell you to thank them before you do... And she's not wrong.

Actually Look at an Actual Thing

This is a hard guideline - so hard that there's a whole later chapter devoted to it. You will learn ***by far*** the most if you let yourself be surprised by the real world and marvel at its complexity.

It's easy to think we already know everything. Staring at the world is the fastest way to remember that we don't.

Later in this book, there's a lot more about ***why*** this is important and ***why*** you learn so much more.

It's okay to look at the real-world inspiration, take notes and then code it later. But you'll learn much less if you code purely from memory. You probably think that you remember things pretty well -- I certainly think I do. If you think so, it's time to sit down with a paper and pencil and try drawing an apple. Then, go look at an actual apple. See the difference? That's the difference between what you remember and what's actually in the real world. Pretty stark, isn't it?

The world is too complicated for you to keep in your head. Great! Go look at the world, be surprised by something and then code it. That's how the good stuff happens.

Break Rules

I don't specifically mean *these* rules, though you can. I mean that you're doing exercises to learn. If there are rules or best practices ("an object should have one responsibility," "don't copy and paste code") it can be very educational to *break* those rules and find out what happens. Sometimes it's even more educational to break them *a lot* and see what happens.

You're also not required to save all the learning until the end of the exercise. If you break rules and it's totally unworkable and you "fail" the exercise... Then you learned something, so you didn't really fail. You just finished early. Well done.

This isn't a test that you administer to yourself. It's a chance to play.

Choosing Task and Purpose

Now you know how to do a single coding study. But how do you use coding studies strategically to gain skill? And what do you do if you're having trouble coming up with a task or a purpose?

Choosing a Task: the Easy Way

There's a way you already know to choose a task, and it's fine. It's not amazing, but it works in a pinch.

You can Google an existing coding exercise. You'll get something, and it may be great, but more likely it will be only okay. That's fine - it's familiar, it's easy, and it will get you practicing. Practicing is better than not practicing, and you may be able to salvage a lacklustre task with an excellent purpose, or with a change of programming paradigm or with a new tool.

Choosing a Task: Do It Again

Rather than Googling an existing coding exercise, you can adapt an exercise from this book or do an interesting exercise you've done before. Keep in mind that re-using a task can be great if the task has some flexibility in the design. You can come back to something like the Merry-Go-Round example as often as you come up with a new feature of kids or Merry-Go-Rounds - and they're both very interesting, with a lot of good choices.

In general, a little simulation like that will often reward coming back to add a new feature or change one out.

Didn't I say earlier that you should throw away the results, not keep them around or add to them? I did. But in this case, you're trying to figure out the fastest way to learn. You're not compromising your learning by trying to turn a coding exercise into a polished, finished product. Instead, you're deciding the fastest way to get to the part of the exercise you care about. Starting from a previous exercise can be a great choice.

Or you can start fresh. There's never anything wrong with rebuilding an old exercise, especially if you're coming to it with fresh eyes. If it's been awhile since you built the old one, it may be useful to build it again. If you've learned something interesting since then, build it again and see if it changes how you come at the problem.

Do you notice how professional engineers often talk about code re-use and DRY ("Don't Repeat Yourself") as a major virtues, but artists and students both constantly rebuild? That's because they're doing different things. If you're building a product for customers to use, by all means use the most bulletproof code you can get your hands on. Re-using is better than rebuilding, and buying is better than re-using.

But if you're trying to learn, rebuilding the same code over and over is often a powerful technique. It shows you how you've changed, and it shows you other approaches you could take to the same problem.

Choosing a Task: the New Tool

Sometimes you want to learn a new tool. In that case, pick a task you already know well with your *existing* tools. What's something small you've done in Ruby (if you already know Ruby well) that you can do in Java? What's something you've done with HTTPClient (if you know it well) that you can use to try out HTTParty?

If you don't have a small task in mind already, consider taking code you've already worked on and modifying from one tool to another. This works especially well with libraries, of course. When I say "plan to throw the result away," that doesn't mean you have to *start* from nothing.

Should you still throw the result away if you're trying out a new library and it might be better? Probably, yes. If you're learning an *interesting* new tool, it should be significantly different than the old one. You should try using it to its limits so that you can find out what those limits are. Instead of holding back and just doing what you have before, you should be playing with whatever's different.

That's great for learning! It's also *not* a great way to get the best final resulting code. In this book, I'm not telling you how to write finished production code. I'm telling you how to learn the most. To learn the most, plan to throw the result away - and use whatever the newest, weirdest features of the new tool are so that you can learn them.

Choosing a Task: the Artistic Way

I suggested earlier that you should look at a real-world system as you build. The easiest way to do that is to choose a real-world system as the original inspiration for what you design and build.

Your task is ***what*** you're building. So how do you abstract a tree or a house or a piece of plumbing into a specification for code?

The simplest way is the most literal: build a simulation. What do these things ***do*** over time? A tree may grow upward and branch. A house can be designed over time (adding rooms? slowly decaying?) or have people move through it. A piece of plumbing may have water flow through, or make odd sounds, or vibrate at a particular frequency - pick something that interests you about the system and simulate that part.

The obvious way to build a simulation is "step by step, forward in time" but there's no reason you have to. Your unit of time could be a second, a day or a year... Or it could be "until the next room is built onto the house."

Or you could simulate over something other than time. You could branch out ("grow") plumbing pipes until they collide with each other, even though they don't literally ***move*** in the real world. You're looking at a system that interests you and figuring out a system. There's no reason that has to be a simulation forward in time.

You might say, "I don't really know how to do that." Great! You have a few example simulations in this book. And you'll learn a lot by trying to do something difficult, even if you "fail" - that is, even if you don't produce something simple and clean that does roughly what you were looking at. Trying to structure a genuinely unknown task is hard. It's so hard that you're likely to get substantially better at designing software if you try it for awhile.

You may have an easier time if you research other ways that other people have built simulations - there are standard techniques for these things, depending how you want to represent the simulations. But keep in mind that your goal doesn't have to be about accuracy. You're playing and exploring, and with luck you'll keep doing so.

You can also start from something like the Merry-Go-Round example. While that's technically a simplified Euler's Method, it's also ***so*** simplified that you can probably learn all you need about the approach just by looking at it.

Here are some starting points for that research, if you want to do it:

- Euler's Method, an easy way to simulate with numbers and derivatives
- Conway's Game of Life, a grid simulation like the "mushrooms" example
- Cellular automata, a complicated generalisation from the grid

I'm also a fan of several similar things, such as Nicky Case's "Simulating the World in Emoji," a gorgeous illustration of a similar concept: https://ncase.me/simulating/

Choosing a Purpose: Who's Your Hero?

In addition to choosing a task, you should choose a Purpose - that is, what you hope to learn from the exercise. How do you do that?

Well, one way is to pick a skill you wish you had and work toward it. Do you love Sandi Metz? She's an easy choice because she *tells you* a lot of how to become more like her. Speakers are great that way - you can watch one of their talks and then do an exercise on what they speak about.