How to quickly and effectively read other people's code

by Alex Coleman | Learning (https://selftaughtcoders.com/learning/), Web Development (https://selftaughtcoders.com/web-development/)

Just the other day, a fellow STCer (Self-Taught Coder) asked me the following question:

> "How do you go about understanding someone else's code? I am starting to feel more comfortable with my own code, but whenever I try to look at something someone else wrote I feel totally lost. I know some of this is unavoidable, especially if the code is poorly (or not at all) documented, but right now I have no strategy at all. Any tips would be greatly appreciated!"

I love this question for a few reasons:

1. The method I'll recommend for reading and understanding someone else's code will also help you: 1) better understand *your own* code; and 2) help you increase the speed and ease with which you understand *all* new pieces of code you approach.
2. It sheds light on one of the most important aspects of learning a new skill, like programming: exposure to high quantity, high quality examples of expertise.

There are a lot of wins here. Let's start at the beginning.

What's the best way to read and understand someone else's code?

The best way I've ever discovered to read and understand someone else's code is to:

1. Find one thing you know the code does, and trace those actions backward, starting at the end.

Say, for example, you know that the code you're viewing ultimately creates a file with a list of movie titles. Figure out where in the code — the *specific, few lines* — it generates that file.

Then, move one step backward and figure out how it places the info in the file.

Then, move another step backward and figure out where the info came from.

And so on...

Let's call those connected pieces of code a "chain of actions."

Inevitably, using this approach will lead you through a bunch of different areas of the code. And that will probably give you a good deal of insight into things such as:

- how the body of code is organized (where variables are defined, where different types of functions are located, etc.)
- the person's style of coding
- how the person who wrote the code thinks and problem solves (this is harder to describe, but it comes intuitively the more examples you see)

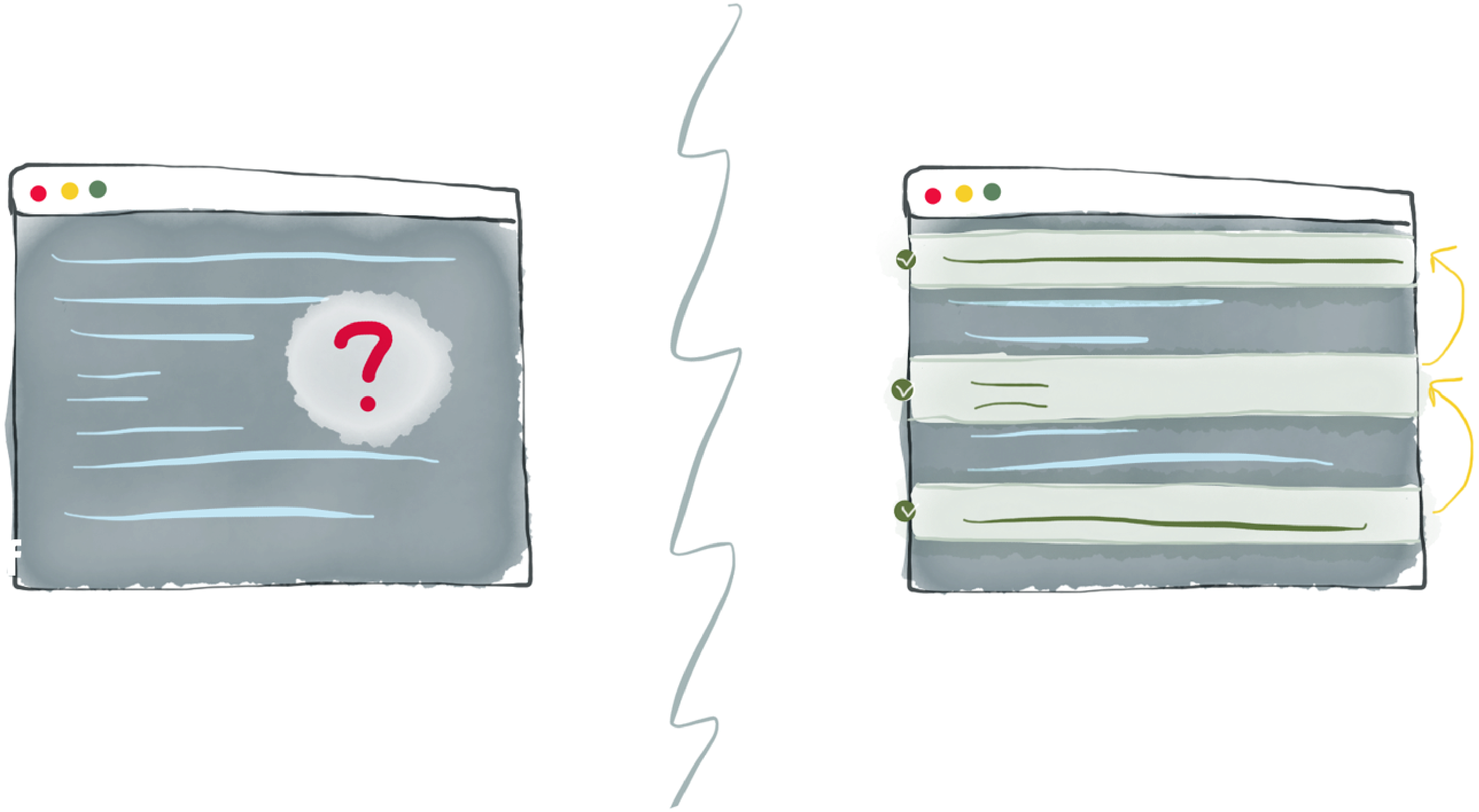
And by doing that, **you'll gradually begin to understand more and more of that full body of code**. So where you started with:

[a big file of code that doesn't really mean much at all to you]

you'll now be looking at:

[still a big file of code, but where you now understand a few specific sections]

It's almost as if you were originally standing in a room that was pitch-black, and, one at a time, different lights throughout the room were turned on to gradually reveal more details of the room's appearance.



Using “chains of actions” to gradually understand a never-before-seen piece of code

2. Rinse and repeat.

Repeat that process multiple times, and you’ll rapidly increase your understanding of more and more pieces of the overall codebase.

Just as parts of the pitch-black room are gradually illuminated, parts of the code gradually “light up” for you, as you understand how they function.

The reason that works well is that, in all cases, **a body of code is designed to tackle one (or more) complex problems**. So you'll always have those "chains of actions" throughout.

And **the more you can gain an understanding of how different parts of the code are connected, the more you'll develop an understanding of the entire codebase, as a whole**.

And, over time, **the more (good) code you see, the easier it becomes to read and understand *all* code, and the faster you can do so.**

..which leads directly into the second reason I love this question: it highlights the importance of exposure to high quantity, high quality examples of expertise.

In programming, **"high quality examples of expertise" = good code** that other programmers wrote.

The importance of exposing yourself to high quantity, high quality examples of expertise

In her incredibly poignant new book, *Badass: Making Users Awesome (http://www.amazon.com/gp/product/1491919019/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=1491919019&linkCode=as2&tag=alepcol-20&linkId=THNHYVSS752ZYNNU)*, Kathy Sierra states that **exposure to high quantity, high quality examples of expertise is one of the two main factors that dictate how quickly and effectively people learn new skills**. (The other is deliberate practice.)

> "The more you watch (or listen) to expert examples, the better you can become. The *less* exposure you have to experts or results of expert work, the less likely you are to develop expert skills."

Let's take a look at the first example that clearly pops into everyone's mind: chicken sexing. Kidding. But it is, in fact, a great demonstration of this concept.

Learning from chicken sexing (yeah, you read that right)

What is chicken sexing? And what does it have to do with exposure to high quantity/quality examples of expertise? Kathy explains:

> "Determining the gender of a newborn chick is notoriously tough, but for large commercial chicken farms, the sooner the females are separated from the males, the sooner they can be on the feeding-for-egg-production path. In the early 1900s, the Japanese developed a particular method for chick sexing and a few experts (reliable, accurate chick-sexers) emerged. Great, we'll have those experts teach others, right? Just one problem: when questioned, the chick-sexing experts didn't know exactly *how* they did it. 'I just knew.'"

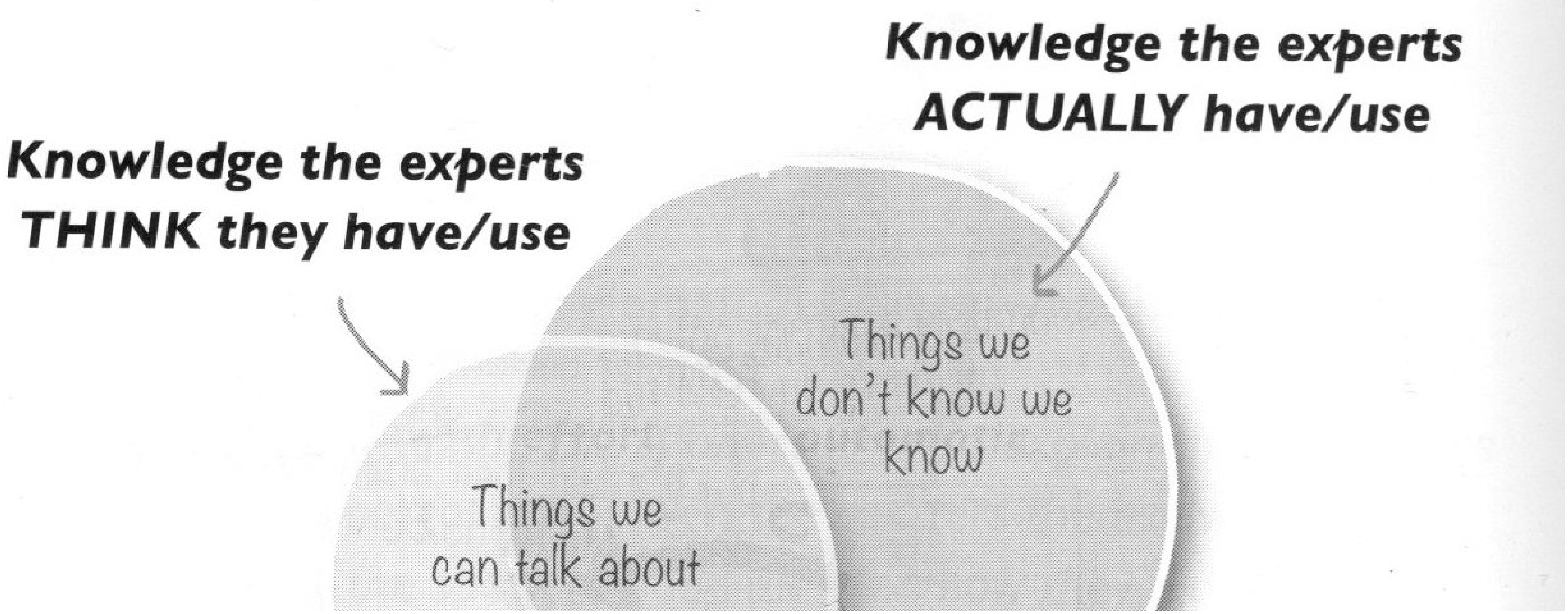
So how do they even go about the training? Well, assume you're one of the new chick-sexer recruits. You're are placed in front of a bin full of baby chicks. Problem is: they all look *exactly* the same to you. But you're told to just pick one up and make a guess as to its sex. As far as you're concerned, your guesses are *completely random*. But Sierra continues:

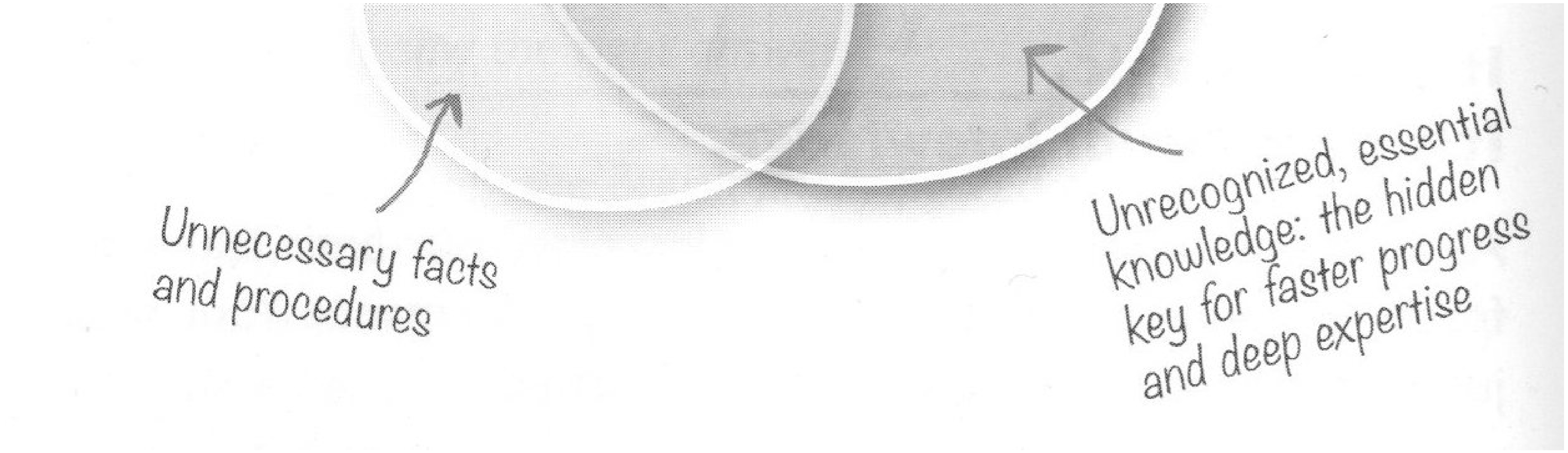
> "After each wild, random, totally made-up guess, the master chick-sexer gives you *feedback*. Yes, no, no, yes. You still have no idea how the expert "knows," but you just keep doing this, over and over.
> And then, eventually, something happens. You begin scoring better than random. You get better. Over time, *much* better. *But you don't know why.* For all you know, you're *still* just guessing, but now it's as if some "mysterious" force is guiding your hand toward the correct bin.

How does all this "magic" work? Sierra brings it on home:

“After enough exposure with feedback, your brain [begins] detecting patterns and underlying structures, without your conscious awareness. With *more* exposure, your brain [fine-tunes] its perception and eventually [figures] out what *really* [matters]. Your brain [is] making finer distinctions and sorting signal from noise even if *you* [can’t] explain how.
Perceptual knowledge includes what we think of as expert *intuition*. The ability to instantly know *which* chess move to make. Or that *this* painting is a *forgery*. Or that *this* house fire will *explode*. Or that there’s *something* wrong with that code, even though you can’t always articulate *how* you know.”

Experts in all domains develop and use unconscious perceptual knowledge


Knowledge the experts ACTUALLY have/use
Knowledge the experts THINK they have/use
Things we don't know we know
Things we can talk about


Unnecessary facts and procedures
Unrecognized, essential knowledge: the hidden key for faster progress and deep expertise

How experts use unconscious perceptual knowledge

from pp. 134 of Kathy Sierra's book, "Badass: Making Users Awesome"

How does that play out in programming?

ost importantly, know that **the longer you're programming — and thus the more code samples you see, of all different kinds — the easier it gets to understand other people's code. And the faster you're able to do it.**

It's a wonderfully self-perpetuating cycle: you read more code; you gain the ability to understand it quicker and more effectively; so you are le to consume even *more* code; and so on.

nd it doesn't stop there: you'll also see huge positive gains in *your own* coding. How so?

1. You'll be able to more quickly understand code samples and examples you inevitably reference during your own programming (e.g. something from an online course; or a snippet from a StackOverflow post).
2. You'll be able to understand past code you've written at a glance. (And, inevitably, down the road, you'll be working with a lot of different pieces of code all together, so this ability pays off big time.)

Ultimately, that translates to:

1. Less pauses
2. More progress

which = **more fun and more enjoyable**. Win!

And you better believe that...

I’ve incorporated all of this into my online course, From Idea To Launch

From Idea To Launch (https://selftaughtcoders.com/from-idea-to-launch/) walks you through building your very own Laravel PHP web application – step by step – based on *your own* idea. And the course:

1. **includes plenty of real-world, expert code examples alongside each lesson**, to ensure you’re consistently exposed to high quality examples of code.
2. **helps you connect with other programmers**, furthering your exposure to examples of others’ code, techniques, and thought processes.

I always aim to practice what I preach. Be sure to read more about the course (https://selftaughtcoders.com/from-idea-to-launch/) if you’re interested in building and launching your own web app.

Shoutout: Thanks to Carol, one of our own, for the inspiration for this post! It’s the best when I get to write about things that come from directly within our group.

> **Alex Coleman** *helps others learn to build web applications with Laravel. His articles and courses have helped over 10,000 developers level-up their PHP web development skills and learn to build and launch their own web applications to the world. If you enjoyed this article, then join his free newsletter (https://selftaughtcoders.com/newsletter).*