In 2024, I’ll have been a programmer for 40 years. I’m not quite there yet, but I’ll get there. That’s different from making the most money developing software, or starting a software company. I do make good money. But that’s not what all this is about.

I think you might want to be a software developer for a long time, in the same way that some people are musicians for a long time, or artists for a long time, or roofers for a long time. If not, you can hit “back” in your browser. It’s cool, no harm no foul. But I think maybe you would like to be a twenty-year programmer, or forty-year, or more.

I’m not going to tell you what language or framework to learn. If you’re good at the basics you can learn anything you want. Until then, you can be bad at whichever one you want. That’s what I did, anyway.

Instead I’ll talk about things that didn’t make sense to me when I started, and now they’re the guiding stars in my sky. That means it’s not about specific technologies. Technologies come and go. Languages come and go. They can’t be your sky.

Let’s start with this: software is young.

Software Development is Young

The computer language Fortran dates back to 1957. There are other languages about the same age (LISP and Algol: 1958, COBOL: 1959) and a few weird contenders for being older (Konrad Zuse’s Plankalkül, 1942-ish.)

Let’s say there have been programmers since roughly 1957. 65 years. I’ve been a full-time paid programmer since 1998 (24 years) and a programmer at all since 1984 (38 years.) I’m pretty experienced. Alan Kay, inventor of the Smalltalk programming language, starting somewhere around 1963 — 59 years. He’s one of the longest-active I’ve found.

By contrast, Yo Yo Ma is 66, and has been playing the cello since he was 4 ½. He is nowhere near being the longest-playing master musician in the world. The Guiness Book says Kasper “Stranger” Malone had professional musical releases over eight consecutive decades. And that’s just professional musicians and songs sold for money. It doesn’t count live performances or anything they don’t charge for. So let’s say a really experienced musician has been working for 15 years longer than our whole discipline has existed.

Our field is young.

You Don’t Have to Start Young

So if I started at 8 and Yo Yo Ma started at 4 ½, is it too late for you? Nah.

I started playing the piano seriously at 45, around a year ago. I think I’m really coming along. If I can keep at it, I’ll be freaking fantastic when I’m 60. I can play some songs decently already. You just get better faster when you already have some background and some perspective, you know? You’ve been practicing picking up skills for a few decades, or more than a few.

Say you’re 50. You could have 30 good years as a software developer, give or take. And if you start at 50, you’re going to be a lot better at 60, ten years in, than I was at 18, ten years in. A lot better.

I meet excellent programmers who started at 20, 30 or even 40. I don’t know any reason you couldn’t start at 50 or 60. It takes time and work. But you don’t have to be young.

Does it mean you won’t be as good, though? Kind of. I won’t be as good on the piano as Yo Yo Ma is on the cello. I won’t be as good as Alan Kay at programming, even though I started younger. You know who else isn’t as good as Yo Yo Ma? Basically everybody else who plays the cello. They don’t let it stop them. A lot of them are good at cello anyway. And I enjoy playing the piano, even if I’m never top-ten in the world.

I still enjoy writing software. My odds of being the next Alan Kay are basically zero. He had already written Smalltalk at my age. “Be okay not being Alan Kay” is necessary if you work in software. Even young Alan Kay wasn’t current Alan Kay. He had to be okay with that.

Order Isn’t Make-or-Break

If you want a long programming career and you’ve just started, I often say “just write some software, any software, it doesn’t much matter what.” You’re going to need to study a lot of different things. It’s not a huge deal what order you study them in.

In 40 years, lots of trends come and go. Desktop apps? Done it. Mobile? Written it. OS and systems? Yup. Web programming? Totally. Your own list will be different, but it will span many kinds of programming. It doesn’t matter much what order you do them, as long as you try different things.

“Try different things” is the key, of course. You’ll get a lot better from ten years each of four different areas than you will from 40 years writing network-socket servers. It’s not that 40 years of depth has no value. Depth definitely has value. So you shouldn’t do one year each of forty different things.

But you have to be careful not to get rigid. It turns out that nearly any discipline has something to teach you. If you’re too stiff to learn it, you lose out.

Try different kinds of programming. Try them in any order. But you should be working on something and getting better.

It’s Early Yet

What if the thing you study is useless? Eh, useless is relative. I put years of spare time into an old MUD programming language called DGD. It certainly wasn’t for the practical value. Almost everything about it was weird and nonstandard, and very little of it was really practical. Yet it taught me a huge amount because it was weird. It taught me things that Ruby on Rails later used, and it taught me things about programming with databases even though it didn’t use a database! It taught me things that were useful in five or six different later languages I later learned. It gave me war stories to tell for hour after hour.

Amusingly, I got a consulting gig in DGD many years later. There haven’t been many DGD jobs in the world, ever. But I had one of them! That’s more practical than a lot of “practical” languages I’ve learned. You never know.

Pretend that Yo Yo Ma was 15, about ten years into his music career. Pretend he’s fussing to you about whether he should play a complicated piece of classical music or fool around with something popular and jazzy. You could examine the merits: the classical music piece will improve his technique, the popular jazzy piece will give him breadth, and so on.

But the real answer is simpler: both have value so pick something and run with it. And if you keep it up for 30 more years, it’s not so important which one you pick. Both can make you better. The important thing is to work. Put real work into either one, and it will be valuable experience. Try to “skate by” without thinking about it, and either one won’t be worth your time.

That’s the answer for young Yo Yo Ma. It’s also the answer for you.

I often say to myself “it’s early yet.” What I mean is, I’m 46. I’m not 96 and done. I have at least twenty more years of this left, and maybe fifty years. So if I do something weird and off-the-wall and it teaches me fun stuff, that’s really good. If I do something practical, that carries me through the short term, that’s also good.

But the important thing is: keep working. It’s going to take a lot of work to make it to twenty or forty or seventy years as a programmer. If two pieces of work are hard to choose between… probably they’re both good, and either would be a good choice. Otherwise they’re both bad and you should find something worthwhile instead.

It’s early yet. Study something interesting or useful. Give it time to pay dividends. But you’re guessing about payoffs that are ten or twenty or thirty years out. Don’t always pick the thing that will be best in eighteen months. You can’t really see what the future will hold.

The Work is Good

You got into writing software because something about it appealed to you. Some part of it was good, or you wouldn’t have bothered.

Try to figure out what that was. What did you like? What drew you in? What’s good about it? It won’t be the same for every person. Not at all.

I love to feel accomplished and clever. That’s not the best thing to do at my job, but I do it a lot at home. I don’t need everybody to see it. I can just be clever by myself. And then, the work is good. I also love to do something that’s different from my job, even if I’m writing code both places. I love to write something other programmers can use and appreciate.

Your reasons will be different. Good. Find them.

The work is good. If it stops being good, you’ll stop too. If it stops being good, that’s an emergency: you need to take a vacation, or reconnect with what you liked, or something. If your work stops feeling good, you’ll burn out fast and hard.

The work is good. If it isn’t, change your work.

None of this necessarily means the job is good, or to change your job. But you need enough good in your work to keep you going. To keep going is a requirement, one way or another. So that may affect your job, too.

It’s not a Sprint or a Marathon, It’s a Diary

It’s easy to decide, “I’m going to be a computer programmer” and lay out an eight-section plan with fifty six bullet points about how you’re gonna do it. I won’t tell you not to get worked up. If you can’t get worked up, why bother?

But I’ll say: don’t take the plan too seriously.

As a kid, I lived 4.2 miles outside a cruddy little town I hated where everybody also hated me. For good reasons, to be fair — I was a miserable little asshole. They weren’t missing out. A degree from Carnegie Mellon got me out of there. Computers seemed magical before I ever saw one, and then magical when I got to play around with an old Apple IIe. At CMU I constantly hung out in the computer labs, because computers were awesome and I didn’t have any social skills.

So: I took a very direct route to writing software. I started early, I worked constantly at it.

But even a guy like me can’t just live or die by software. Eventually I found friends and wanted to keep them. Eventually I got tired of being lousy to talk to. Eventually I wanted, and found, a wife. Eventually I had kids and tried to do well by them. Eventually I discovered that emotional rigidity and suppression only get you so far, even with a good job.

At some point, you stop calculating because you can’t do everything by calculation and planning. At some point you’re not “off task,” you’re just “living your life.”

So if you beat yourself up for doing stuff that’s not software… Maybe don’t. Why do I talk about music so much here? Because I’m learning that too, not just putting everything into software. Art, too: I gave a talk and wrote a book about stealing practice methods for software from artists, and I’m illustrating more and more of my own stuff. Obviously I also speak and write. I get broader in my interests every year.

It’s not a failing. It’s not giving up. You can’t predict what’s going to be valuable, so you should learn some of everything. That’s the same thing I say about what kind of software you should write, but broader. And that’s the lesson: the longer you live and the better you work, the more you realise that everything (and everybody) has something useful to teach you.

Just keep working.

Learning to write software isn’t a task you plan out: get these bullet points done and you’re a programmer. As soon as you write your first program, you’re a programmer. If you’re reading this article, odds are that you’re a programmer. If not, go write your first program in an online code learning site — it should take you under 20 minutes.

Sometimes you’ll want to improve at something in particular: your programs get snarled up together as they get bigger, so you learn some modularity and encapsulation. You’re trying to handle thousands of combinations with good performance, so you maybe pick up some discrete math. All this stuff is available online, usually with references to books and classes if you want to go deeper.

Learning non-software stuff is the same way. The guy who wrote the game Chicory coordinated his onscreen sounds with the key he had written that part of the game music: you get harmonious notes because it plays slightly different chimes, depending on the song in the background. How freaking cool is that? And I realised I didn’t know nearly enough to even appreciate it. And it helped push me toward music last year.

Just keep working. There’s infinite interesting work to make you better. Don’t be afraid of work that looks impractical. I don’t know what I’m going to do with music for my software work. Maybe nothing. Maybe everything. The future is very far away, and I’m not good at predicting what will be helpful in twenty years.

None of us are.

You’re not running a sprint or running a marathon. Instead, you’re keeping a diary. In ten years, will you page back through it and say, “wow, I did some cool stuff?” Or “huh, I was a pretty interesting person?” I guess you could keep a diary that says “I was pretty good at Java syntax.”

But why?

Don’t Confuse Work and Career

Don’t I want to retire at some point? That would be nice. But I’m not going to stop working. I will stop doing work I don’t like for money, especially if it doesn’t teach me anything awesome. My current gig working on YJIT is full of code I’d write for free… But if I wasn’t being paid I’d never touch an expense report and rarely write status updates. They’d get a lot less sysadmin work or Git history pruning. Don’t confuse work with your career. They’re not the same thing. They’re only barely related. Writing software is incredible work. It’s an okay-or-better career.

That’s another reason I keep talking about musicians. There are people who get paid to write and play music. But many (most?) musicians never get paid and still do it. The work is interesting, powerful and satisfying. If you can get paid for it, that might open up a lot more time for you to work. But the work is about the work, and the job is about clearing the way for you to work. To me, writing software is like that. Maybe for you too?

The most important thing, though, is to tell which kind of advice you’re getting at any given time. You’ll get advice about the work. You’ll get advice about the job. If you mistake one for the other, the advice won’t make much sense. To make it extra annoying, people will use the words “work” and “job” and “career” and so on interchangeably. So just listening for the right word won’t tell you which is which.

There are No Defined Levels

If the order you learn doesn’t matter, then there’s not really a such thing as “level one” or “level two.” Or any other levels. You’ll get different recommendations about what language or tech to learn first, and that’s fine. But if you forge your own path, that doesn’t mean that you didn’t do the basics and you suck. Eventually, if something is important, you’ll figure out that you need it, and you’ll get back to it.

Or you won’t. But mostly, you will.

You hear scary stories about somebody neglecting something basic and just being awful, year after year. And that happens. Like, you could write fifteen years of horrifying spaghetti code where every module reaches into every other module to directly set each other’s variables. And if you keep trying, you will either figure out why that’s bad, or you will invent a new style with a new solution to that problem, and get good at something weird. One programmer’s “new paradigm” is another programmer’s “something weird,” always.

The problem is when you don’t keep trying to get better. If you keep making new mistakes, you’ll learn things that are useful, or beautiful, or bizarre. If you keep doing the same thing without trying to improve then sure, you’ll keep sucking.

But that’s true even if you go through somebody’s pre-designed levels in order.

Bruce Lee said, “I do not fear the man who has practiced ten thousand kicks, once each. But the man who has practiced one kick, ten thousand times, him I fear.”

The trick is to care and to improve. You can do that while focusing on one thing, or ten things, or even a hundred things if you keep going long enough. You probably can’t manage ten thousand things. But if you really try hard enough, you could prove me wrong. You’d have to do something weird.

Which is good. New paradigms – pronounced as “something weird” – is how we collectively get good at new things. If “do something completely new every Tuesday” is your one kick, then just keep on kicking. You’ll figure out it’s bad, or you’ll get really good at it.

The Better You Get, the Less You Look Like Everybody Else

Early-career training (code schools, blog posts, university classes, books) often feel kind of like an assembly line. There are a bunch of really basic skills — e.g. writing functions, debugging, estimation, talking to your team — and they’re trying to make sure you have a basic level of competence in each one.

It would be easy to think that means that for, say, a Principal Engineer, there’s a list of skills you need, but there are more skills on the list and the skill level has to be higher. But that’s not true. Not at all.

It’s not just job levels. It’s also ‘levels’ in open-source, levels of respect.

You can become massively respected by taking a fairly simple piece of code and writing (in English, to humans) about it in massive detail, like Patrick McKenzie with Bingo Card Creator. You can become respected by writing something really profitable, like a company founder. Or writing something deep and complicated, like an obscure language (Haskell, anyone?) These paths have almost nothing in common with each other beyond basic competence.

You need to be really good at something, and that something needs to be popular or profitable or otherwise going to ‘make it big’ somehow. This should sound very vague, because it is. It’s also subjective. If you set out to make Bill-Gates-level software money and you wound up writing Haskell (complex, deep, critically acclaimed, not that lucrative,) you’d be a big failure, and vice-versa. And the relevant skills are completely different.

This is why it’s silly to ask a question like, “I’m a software engineer with fifteen years of experience, what’s the usual salary for that?” Fifteen years of experience is so much that you should be almost completely unlike an engineer standing next to you who also has fifteen years of experience. Did you write a book? Work on a massive project that made money? Put together an interesting open-source project? What did you do with that fifteen years? It’s like saying, “I’m a musician with twenty years of experience, what should I charge per hour?” There is no simple answer to that question, and there shouldn’t be.

It’s not just about salary, either. You could ask, “I’m a software engineer with fifteen years of experience, that means I’m competent to lead this project, right?” The answer, of course, is “maybe.” The followup question is “what did you do in that fifteen years?”

You Learn Deep Principles by Doing the Basics

I don’t tell people to go out and learn the deep principles of software design at first. That’s because if you try to learn them as theory, without practical experience, you’re almost guaranteed to do it wrong. First learn to build a usable piece of software in some actual language. I don’t really care which one. Get some genuine mistakes under your belt. Then we can talk about what theory exists to fix the problems you personally have.

If it turns out that you’re fantastic at modularity, then great! We can talk about other problems that you do have. Works for me. If you have no problems but the result is still simple and bland… then you do have problems, and we can talk about that. If you have no problems that you care about, that just means you’re fine building this same level of software for awhile. Up to you. When you start to feel like you’re getting stale, then you do have a problem, so you can look for a solution. Until then, do what works for you.

Then, do that same cycle for twenty-plus years. Build, make mistakes, learn some theory, fix your mistakes. I don’t much care about what order you do it all in.

Does that mean if you learn theory first you’ll “corrupt” yourself and be bad forever? Nah. But it’ll take awhile before you can use what you learned properly. So it’s not the fastest way to build good software. In the long run, it’s all fine. It’s early yet, right?

What Tech You Use Matters

I’ve spent a lot of time telling you that what tech you learn in what order doesn’t matter.

When does it matter?

It matters that you learn some variety. If you always do one thing, you don’t know what bad habits you’re picking up. If you always do on-just-one-machine coding, you don’t know how much you never learned about networking. And eventually networking is probably important. If you only work on web servers and never on mobile or app code, you never learn why “works on my machine” is so bad. You can do one thing for quite awhile — five years? Ten years? But if you’re going to make it to forty years, you need to learn about a lot of other tech that is touched by your software on the edges. You need a bit of perspective.

Perspective is like other deep principles — you get it by doing work and seeing real-world problems. And you get it through a variety of work, so that you see more than one point of view. You realise you have a perspective problem when you say “well, my part works,” but the whole system is failing because other people’s part isn’t working. You learn to care about something bigger than just your own code. And that turns into software architecture, but it also turns into perspective and empathy. A real-world software system that does something useful has lots of software, but it also has lots of people. You can’t just ignore the people.

And that means learning a variety of technologies, and a variety of non-technology skills.

The other reason it matters what tech you use is that some languages or libraries will make you better at specific skills. You “should learn at least one functional programming language” in the same way you “should learn to play a piano piece by Mozart.” It will teach you something interesting and make you better. If you want to make it forty years in programming, you’re going to need solid fundamental skills.

Can’t you learn solid fundamentals from any tech, not just the weird hard ones? Sure. And if you have done that, then learning the software equivalent of a Mozart piano piece won’t be hard for you. There’s nothing magic about Mozart. He tends to write hard music where you have to move your fingers quickly and accurately. There’s nothing magic about Haskell. It needs you to be careful and exact about modularity and dependencies. If you’re already fantastic at those, you’ll do great.

But if you’re lying to yourself about being fantastic, you will discover that you’re lying.

The best thing about my breezy “do whatever feels good” approach to learning software is that whatever you’re doing, you can mostly keep doing it. The worst thing is that I’m not holding you accountable for your screwups. I’m just assuming that you’ll handle that.

That may be great. It may be awful. You get to choose. I’m not your mother or your mentor.

Most of us need the occasional kick in the butt to notice our problems. I certainly do. One way I kick my own butt is to learn something hard and exact, where if I make a mistake I’ll notice it.

Look to Other Fields, Learn From Other Fields

If our industry is young, what does that mean? It means we’re still figuring out the basics.

When I was in university, from 1993 to 1998, test-first and test-driven development weren’t really a thing. Agile wasn’t really a thing. The approaches existed, but weren’t popular or well known or well developed. Source control existed but wasn’t good and wasn’t widely used. Open source existed, but it was generally assumed that open source meant cruddy software. The idea that Linux was, or could ever be, the best server OS was a weird thing that a few zealots maybe believed. It wasn’t well settled that C was fast enough to replace assembly language for speed-intensive applications. “GOTO” was, finally, understood to be bad and we should avoid it. The web was just starting out, and it started supporting pictures on web pages partway through my school time.

Things change fairly fast in our industry. To put the same thing another way, we’re still incompetent at the basics.

You can learn a huge amount from other fields. I wrote a book on how to steal artists’ practice methods. I keep talking about musicians here. Art and music are old disciplines. We have been teaching them a long time. Art and music teachers are better than computer science professors. They have thousands of years of head start.

So if you have a problem and you set off to learn the best way computer programmers have handled it… you’ve probably limited your options massively. It’s better to think about how anybody handles the problem. The Checklist Manifesto by Atul Gawande talks about the very different ways that pilots and skyscraper builders and doctors handle checklists. They are all great approaches. In software, we’ve just barely started to steal a few of any of their tricks. So if you learn from software engineers, you’ve already skipped learning the best ways to approach most checklist-shaped problems.

Checklists aren’t unique here. Jazz musicians collaborate at a level programmers can only envy. Serious visual designers produce results that data scientists can try to mimic, but we’re not good at it. My jaw drops when I watch culinary-school-trained teams and their workspace organisation. Work in a librarian-organised data space for a few years and you’ll understand how badly we toss together our documentation.

If you’re trying to be as good as the very best programmers, you are seriously selling yourself short.

I’m sure you’ve noticed the proud way that programmers tend to ignore non-programmers when it comes to advice. “Mere” QA people and ops folk tend to be treated as second-class citizens and we often act like they have nothing to teach us. And that’s beyond idiotic. If you’re smart, you’ll figure out what they’re good at and why it matters.

If you think the answer is “they’re good at nothing, and it never matters” then you’re never going to be as good as they are at what they do.

It’s hard to pick up skills from very different disciplines. Learning enough art to steal how they practice is hard. It can take years. But the disciplines that are a lot like programming — database administration, say, or DevOps — are close enough that it’s worth trying to understand them.

That also means that if you have a hobby in woodworking or dance or karate, you should maybe think about what skills that gives you. Another programmer would have to spend years to learn it, but you already have. I expect it’s taught you some transferrable skills. I don’t know which ones, since I don’t do woodworking or (mostly) dance or karate. That’s why I’m here talking about art and music, the ones I do know a bit.

Programmers are Contrary

If you learn from other fields, it’ll make you weird. Mostly that’s good, but sometimes it causes friction.

For instance, artists and musicians and writers all know that if you repeat the same activity, over and over, you’ll get good at it. Rewriting is a thing. Practicing the same piece of music repeatedly is a thing. Drawing human figures and painting still-lifes over and over is a thing.

We have a name for it in software, too: “reinventing the wheel.” It’s treated a bad thing, and we do it shamefully and in private. We look for ways to let the computer do all the repetition so that we can do only new work.

In fact, saying that you do the same thing over and over is a fast way to get shamed in public. It costs you reputation. Doing things in a weird way will often cost you reputation. Reputation, of course, is also tied to things like title and money.

This is why most of us are really bad at starting new projects. It’s why most of us can’t talk about when to use one syntactic structure or another, and why we act like there are open-and-shut rules for indentation that should be enforced by a machine, as if there is a simple, correct way to do it. Because the way you build these skills is repetition, and getting better over time, and believing in expressiveness, and communicating with other humans. Things that we stigmatise, as an industry.

Does this mean that you, individually, need to be bad at these things? Not at all.

You aren’t going to argue programmers into thinking in some other way. Maybe a few, individually. But the answer is not to drag everybody into being better. That will take decades or longer. You don’t have that kind of time. You want to get better, faster than that.

But you personally can do things that are “bad practices” but in fact make you better.

You can reinvent the wheel. You can repeatedly write the same thing. You can also write code in “bad” ways and see what happens.

Be careful with best practices. They are like other forms of advice: they mean that somebody else did the work to think through it and get smarter, and you’re just using the simplest thing they came up with.

That’s fine to start out. But it’s terrible as a way to get better.

Understand that if you learn from artists and librarians and sous-chefs, you can’t easily drag the rest of the software world with you. Just because you’re right, doesn’t mean you can convince them. Your way of doing it right might even need skills they don’t have.

But you can get better if you don’t care too much what the people around you think. And you don’t have to tell people how you did it.

The better you get, the less you look like everybody else. Including, and especially, the “everybody else” who are really good at what they do.

For the same reason, be careful with tools and enforcement. They are designed to keep up a minimum level of competence. They are not designed to let you be really good at something unusual.

And you want to get really good at something unusual.

Be Careful About Productivity Tips

People will tell you that the trick to getting good is constant work. I mean, I’ve told you that at least four or five times so far. They’ll say that the trick is to get a little bit better every day and have it multiply.

Here’s the trap in that advice: most of the time, practicing won’t make you significantly better. Every so often it will make you better by a huge jump. Some of the huge jumps will make you worse at several other things.

If you are expecting a constant 20% improvement per year, compounded monthly, this is going to feel awful.

Efficiency only gets you so far. A lot of these processes are enormously inefficient and totally unreliable. Sometimes they’ll give huge gains. Often they’ll give nothing. Occasionally they’ll hurt you more than they help.

That’s what success looks like, for this specific task.

Productivity and efficiency advice, the nice tidy kind, is usually for well-structured tasks that you understand well. A forty-year programming career is not that kind of task. Again: think diary, not marathon or sprint. Trying to keep a progress bar for yourself is going to go poorly.

Over time, you’ll learn to be careful of anything too neat and tidy. Again: advice is expertise with all the most important bits removed. Yes, that includes what I’m saying right now.

Efficiency is always some specific type of efficiency. An approach that is time-efficient may be money-inefficient. Either one may require fewer children than you currently have, or a quieter place to stay. Efficiency is how you refine a strategy that’s already in place and working okay.

I don’t talk much about efficiency tips. I’m not telling you the very fastest ways to do these things. The most carefully-optimised approaches are fragile, basically always. Efficiency is fine. Optimisation is fine. It can help an extra little bit, once you have the basics working.

The hard part is getting the basics working. So that’s most of what I’m telling you. Being efficient afterward is less important and easier.

But Seriously, Mostly Just Do It

These are all perfectly good things for me to say. Most of it is to reassure you, not to change your approach. It’s not a coincidence that I keep recommending advice from nontechnical disciplines, which you already know, rather than forums full of tech nerds, dripping with the intolerant zeal of the recently converted.

If you write programs, you’re a programmer. Or a coder, or a software engineer, or whatever you feel like calling it.

If you keep writing programs, you can be a whatever-number-of-years programmer. Nobody’s policing this stuff. Or at least, if somebody does, you know you can safely ignore them. I mean, somebody’s gonna comment here, “he’s not a forty-year programmer, he’s been full-time less than twenty-five years!” And that’s stupid, but we’re talking about the internet here. Somebody’s gonna say it. Sometimes a title isn’t even meant as self-description, but you’d have to read literally the first sentence to understand that. Two sentences, to get the full context. And JoeRandom\_420 isn’t made of attention, you know.

In any case, if you keep at it, you qualify. This isn’t a formal club where anybody gets to throw you out. If you do the work for awhile, you get to say you did the work for awhile.

How do you do the work for awhile? You persist. You manage your own expectations. You notice when you’re making mistakes and fix them. You try to be careful not to be too rigid. You try to keep enjoying the work.

All that stuff you’ve just seen.

The rest is time, and maybe a bit of staying mellow.

Forty years will go by before you know it. As it says next to an oak tree, on my dad’s bumper sticker: “you, too, will be old soon.”