Building a Career in Software

A Comprehensive Guide to Success in the Software Industry

Daniel Heller

Apress®

BUILDING A CAREER IN SOFTWARE

A COMPREHENSIVE GUIDE TO SUCCESS IN THE SOFTWARE INDUSTRY

Daniel Heller

Apress®

Building a Career in Software: A Comprehensive Guide to Success in the Software Industry

Daniel Heller
Denver, CO, USA

ISBN-13 (pbk): 978-1-4842-6146-0 ISBN-13 (electronic): 978-1-4842-6147-7
https://doi.org/10.1007/978-1-4842-6147-7

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Shiva Ramachandran
Development Editor: Matthew Moodie
Coordinating Editor: Nancy Chen

Cover designed by eStudioCalamar

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 100043. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484261460. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

For my parents.

Contents

About the Author

Daniel Heller is a Staff Software Engineer in Infrastructure at a San Francisco based software company. In earlier lives, he has led reliability efforts on Uber Eats, built monitoring systems at AppDynamics, helped port iOS to the ARM64 architecture at Apple, directed the responses to dozens of high-stakes production outages, and managed teams of up to 25 engineers.

Along the way, the author discovered a love of mentorship and had the good fortune to mentor tens of talented engineers. Those engineers inspired him with their hundreds of questions about career paths, technical trade-offs, and day-to-day effectiveness; when a short blog post on those themes brought a riot of responses about maturing professionals' need for guidance, the author set out to fill the gap with this book.

Acknowledgments

I'm in the debt of Simon Newton, Angie Zhu, and Dave Pacheco for their encouragement to see this project through, their invaluable feedback on my first draft, and everything I've learned from witnessing their excellence.

Thanks as well to my intrepid early readers, Syrie Bianco, Andrew Mains, Carissa Blossom, Adam Cath, Marek Brysa, Dan Simmons, and Courtney Ryan; their feedback made a difference.

Thanks to Prashant Varanasi and Akshay Shah for an impactful nudge to get started at the beginning of the project.

By no means least, thanks to Matt Moodie and Shiva Ramachandran at Apress for their wonderful insight in shaping this mass of text into a book.

Introduction

In the last three years, I've realized that software engineers are starved for guidance about the professional world. I've spent those years working in a large team filled with bright, motivated programmers in the early years of their careers, and gradually, mentorship has come to be a huge part of my job. Most strikingly, engineers have taken me aside again and again and again to ask questions I recognize from the early, stressed-out days of my own career:

- Should I change jobs? Which job should I take?
- How do I grow as a technologist?
- What should I do when I don't agree with the technical decisions on my team?
- How can I make this meeting more effective?
- How should I prepare for my interview?
- How do I get promoted?
- How can I make this email better?
- How do I find a mentor?
- How do I mentor my junior colleague?
- What should I do when I'm on call and I don't know how to handle a problem?
- What areas should I focus on to be a better engineer?
- How do I deal with recruiters?
- …etc., etc., etc., etc., etc.

My colleagues' tremendous appetite for guidance has shown me that there's a critical gap in today's Computer Science education: young software engineers enter the industry with excellent technical preparation, but no one has taught them a darned thing about how to be a professional engineer—they have to teach themselves, and inevitably the hard way.

This book aims to fill that void with a professional manual for the aspiring software engineer, a guide to managers, role changes, professional technical practices, technical communication, meetings, on-call, project management, advancement, ongoing study, mentorship, compensation, and more.

For my part, I'm a software engineer at a major software company. I've been writing code and managing engineers for 12 years—I've worked at Apple, Uber, AppDynamics, and Microsoft (as an intern), managed teams of over 20 people, interviewed literally hundreds of engineers and managers, and been interviewed scores of times. I've written production JavaScript, Java, C++, Go, C, and assembly, shipped code in the web browser and the kernel, and led the responses to perhaps a hundred production outages. And I continue to do those things today; I'm not a consultant or an "architect" but a regular working coder, fixing bugs and debugging outages, trying to solve the toughest problems I can find with my code and my insight, because I enjoy it and think I do it reasonably well.

Most importantly for you, I'm not an especially gifted programmer; respectable definitely, above average on my good days, but I'm nothing like a 10x coder. So, I've made a fun and reasonably remunerative career on everything but coding brilliance—discipline, study, communication, project management, collaboration, prioritization, etc., etc., etc. This book will help you build your career the same way.

Part 1 is about careers: hiring, compensation, and promotion work in tech companies, how to best navigate those processes, and how to chart a course for growth and advancement.

Part 2 is about the sundry nontechnical skills that help you get traction in your daily work: project management, running meetings, working with your boss and peers, recovering from mistakes, team citizenship, and many other subjects I've found to challenge engineers in the workplace.

Part 3 goes deep on the single most important nontechnical skill for programmers: the sadly neglected art of engineering communication. It starts with a holistic model of communicating at work, then moves on to practical treatments of topics like technical writing, email, and asking effective questions.

Finally, Part 4 is technical; it covers a carefully curated selection of technical subjects that I've found particularly difficult for new software engineers—the kinds of issues that come up every day in software offices and never in software classrooms.

This book strives to offer you the best possible returns on your time; it treats a wide range of subjects with short, stand-alone sections friendly to random access as well as cover-to-cover reading. I hope it will arm you with the tools to steer your career with confidence, save you some or all of the mistakes that taught me my lessons, and ultimately help you succeed as a professional in software.

PART

I

Career

CHAPTER

1

The Big Picture

If You Only Take Away One Thing

Here's the most important lesson in this whole book: you need to own your own career, because no one else will guide you. Good mentorship can be wonderful for the .01% of engineers who find it, but in all likelihood, you are going to teach yourself 99% of everything you'll learn as a professional; great projects may fall into your lap once in a blue moon, but more often, you'll have to find your way to them yourself. Therefore, the most important tools in your toolbox are going to be personal responsibility and initiative; those qualities are what make you a trustworthy (and valued) professional, but also how you grow and advance. We'll discuss this principle in many contexts throughout the book.

What Is the Job?

Software engineers design, build, debug, and maintain software systems, which is to say they write text that tells computers to do useful things. At the time of this writing, these skills are some of the most sought-after in the global economy.

This work can take many forms. Some engineers are generalists, with the skills to make changes in almost any system, while some are specialists with profound expertise in one area; some maintain and improve existing systems, while some write new ones from scratch; some move from project to project, getting things working and moving on, while others own and develop one system for years. Some of us work at companies whose main product is

D. Heller, *Building a Career in Software*, https://doi.org/10.1007/978-1-4842-6147-7_1

software, while others work on ancillary systems to help produce a non-software product or service. Day to day, though, the foremost qualities of our work are more or less the same:

- We write a lot of code.
- Almost as much, and sometimes more, we debug code (analyze why things are going wrong).
- We work normal-ish hours (9–5 or 10–6), with extra hours tacked on at more intense companies and an hour lopped off here or there at slower shops.
- Collaboration is a big part of our jobs: we coordinate with other engineers, product managers, customers, operations teams, and etc., etc., etc.
- We write frequently for humans: design proposals, status updates, defect postmortems.

What It Means to Grow

Engineering is the enterprise of building and applying technology to solve problems, and I find joy and comfort in the observation that whatever the pros or cons of any one project, the world needs people who build things. My definition of growth derives from this observation: if we exist to solve problems, then growth is being able to solve more, tougher, and bigger problems. We do so with a vector of skills built over time:

- **Coding**: Clarity, testing, documentation, discipline in scope of diffs.
- **Project management**: Identifying dependencies, updating stakeholders, tracking tasks.
- **Communication**: Clear emails, engaging presentations, evangelizing our ideas.
- **Personal organization and time management**: Not dropping balls, prioritizing effectively.
- **Architecture**: The macroscopic design of systems.
- **Leadership/mentorship** at a level appropriate to their position.
- **Emotional skills**: Empathy, confidence, stress management, work–life balance.

Developing on each of those dimensions is certainly growth. And when we apply those skills successfully, we enjoy four pleasing and necessary benefits:

- Money
- Respect
- Title (bureaucratic blessings)
- Fulfillment, pride, and a sense of purpose

Acquiring each of the above is satisfying and practically beneficial. All of the preceding skills can be dissected in great detail, and much of this book does exactly that. I ask you to remember, though, that everything derives from our essential raison d'être as problem-solvers: the world needs problems solved, so companies need engineers who can solve them, so our impact is the foundation of our career progress.

Ten Principles

I once gave up a team to a new manager. Reflecting on our time together, and thinking about what I'd taught well and poorly as a manager, I wrote a short essay about the most critical practices that I think lift a newly minted software engineer from amateur to seasoned professional: the path from fixing bugs as an "Engineer 1" to leading major projects as a "Senior Engineer."

I was shocked by how strongly people responded to that little list of practices—it seems to be a hard-to-find lesson. It still captures what I see as the most important principles for personal growth and building a successful career, and I'll reproduce it here to set out the principles that thread through the more specific advice later in the book. These are the most important lessons that I wish I had learned years earlier than I did; I sure wish someone had sent it to me when I was 22.

1. **Reason about business value**: Reason like a CEO. Understand the value of your work to your company, and take responsibility for reasoning about quality, feature richness, and speed. Your job isn't just to write code; your job is to make good decisions and help your company succeed, and that requires understanding what really matters.
2. **Unblock yourself**: Learn to never, ever accept being blocked; find a way by persuasion, escalation, or technical creativity. Again, your job isn't just to write the code and wait for everything else to fall into place; your job is to figure out how to create value with your efforts.

3. **Take initiative**: The most common misconception in software is that there are grown-ups out there who are on top of things. Own your team's and company's mission. Don't wait to be told; think about what needs doing and do it or advocate for it. Managers depend on the creativity and intelligence of their engineers, not figuring it all out themselves.
4. **Improve your writing**: Crisp technical writing eases collaboration and greatly improves your ability to persuade, inform, and teach. Remember who your audience is and what they know, write clearly and concisely, and almost always include a tl;dr above the fold.
5. **Own your project management**: Understand the dependency graph for your project, ensure key pieces have owners, write good summaries of plans and status, and proactively inform stakeholders of plans and progress. Practice running meetings! All this enables you to take on much bigger projects and is great preparation for leadership.
6. **Own your education**: Pursue mastery of your craft. Your career should be a journey of constant growth, but no one else will ensure that you grow. Find a way to make learning part of your daily life (even 5 minutes/day); get on mailing lists, find papers and books that are worth reading, and read the manual cover to cover for technologies you work with. Consistency is key; build habits that will keep you growing throughout your career.
7. **Master your tools**: Mastery of editor, debugger, compiler, IDE, database, network tools, and Unix commands is incredibly empowering and likely the best way to increase your development speed. When you encounter a new technology or command, go deeper than you think you have to; you'll learn tricks that will serve you well again and again.
8. **Communicate proactively**: Regular, well-organized communication builds confidence and goodwill in collaborators; knowledge-sharing creates an atmosphere of learning and camaraderie. Share knowledge, and set a regular cadence of informing stakeholders on project goals, progress, and obstacles. Give talks and speak up judiciously in meetings.

9. **Find opportunities to collaborate**: Good collaboration both increases your leverage and improves your visibility in your organization. Advancing your craft as an engineer requires you to have an impact beyond the code you write, and advancing your career requires, to a certain degree, building a personal brand at your company. Cross-functional projects and professional, respectful collaboration are critical to both.
10. **Be professional and reliable**: Think of yourself as a professional, and act like one. Come to meetings on time and prepared, then pay attention. Deliver what you say you will, and communicate proactively when things go wrong (they will). Keep your cool, and express objections respectfully. Show your colleagues respect and appreciation. Minimize your complaining; bring the people around you up, not down. Everyone appreciates a true professional; more importantly, it's the right way to behave.

Your Relationship with Your Employer

Your company is your counterpart in a business transaction where you exchange your valuable skills for their valuable money—your employer is not your mother, your father, or your friend.

Like any firm doing business with another, your expectation should be that your company will make every decision out of rational self-interest. This profound truth has many important corollaries, foremost among them that

- Your company will never do anything for you out of sentiment.
- Your company doesn't owe you education, career development, a raise, or a long-term guarantee of employment.
- Everything your company does is business, not personal, and you shouldn't take it personally.
- You don't owe your company your personal loyalty—they certainly don't see themselves as owing you any.
- When you want something from your employer, you should approach it calmly, as a negotiation between two businesses.

None of these means you mistreat each other: like any two firms doing business, you aim to build a trust that allows for a long-running and mutually fruitful business relationship, and for both of you, building a good name as a trustworthy partner keeps other doors open.

We should approach our relationships with our employers calmly, without a sense of entitlement, aiming to follow our own ethics, firmly represent our interests, and secure the most favorable, mutually beneficial relationship we can. And if we can't get what we want, we shouldn't degrade ourselves by whining—we should sell our skills elsewhere on more favorable terms or accept our situations as the best available.

CHAPTER

2

Landing Jobs

This chapter introduces hiring processes, interviews, and job offers—it aims to demystify the intimidating but mostly predictable journey from the wilderness to a job building software.

Many large tech companies' hiring systems are approximately the same. End to end, the process can take anywhere from < 1 week (for small startups where every stakeholder can get in a room on 5 minutes' notice) to multiple months (Google is famous in Silicon Valley for processes of 4–8 weeks with many stages of committee review). This section will outline the process, with subsequent sections treating each area in detail.

Before we begin, I'll note that smaller firms, especially early startups, often work very differently—they're much more likely to have informal, personality-driven processes, perhaps as simple as a conversation or meal with the team. Coding interviews are also anecdotally less prevalent outside of the United States.

The Recruitment Process

Resume Review and Recruiter Phone Screen

A recruiter screens your resume or LinkedIn profile. If they like what they see, they speak to you on the phone for 20–60 minutes, asking you questions about your interests, experience, and job/salary expectations. The recruiter then makes a decision about whether to pass a candidate on. They do not have technical expertise (though a hiring manager will have given them some

© Daniel Heller 2020
D. Heller, *Building a Career in Software*, https://doi.org/10.1007/978-1-4842-6147-7_2

keywords and context), so their decision is based on imperfect information, even relative to everyone else. Nevertheless, they have considerable discretion in whom to move forward with and whom to drop.

Technical Phone Screen(s)

You do one to two technical phone screens, each 45–60 minutes, with engineering managers and/or engineers. They ask you questions about your experience and likely have you write code in a shared editor like CodePair (or even a Google Doc).

On-site Interviews

You go to a company's office and do four to seven interviews of 45–60 minutes, each with one to three engineers or managers. You write code (either on a whiteboard or on a computer), design systems, and answer questions about your experience and interests. In between, you have lunch with a team.

Take-Home Coding Exercise

Not all companies use this stage. You're given a coding problem to work on for a few days on your own, then send the code to be reviewed by engineers.

Decision

Either a hiring manager or a committee makes the decision about whether to extend an offer. The committee may either be composed of interviewers and a hiring manager or drawn from a central committee (famously the custom at Google); generally, more senior/experienced committee members carry more weight.

The hiring meeting often begins with a simultaneous "thumbs up" or "thumbs down" from each committee member,[1] followed by a discussion to try to reach a consensus on whether to make a hire. The criteria are never objective in the sense of being measurable by a machine—instead, each committee member uses their intuition, sometimes against a written rubric of subjective criteria.

[1]Believed to reduce the risk that people will change their votes silently due to one strong voice; I don't think it does much.

Offer and Negotiation

A company's HR department and hiring manager (or in some cases, an independent committee) craft an offer. The main parameters of the offer are

- Level/title
- Base compensation
- Equity compensation
- Signing bonus and relocation
- Start date

All of these parameters are determined by your experience and interview performance (i.e., the company's perception of how valuable your work will be) and your competing offers, which they may try to match or beat.

Referrals

Companies usually have internal systems for employees to refer others for jobs; you may well be asked to refer others to your company or want to be referred elsewhere.

A referral with a strong personal endorsement is a big deal—it bumps a candidate to the head of the line at the screening stage, and if the referrer is well-regarded, it can make the difference at decision time. A corollary is that you should save your own strong referrals for people you trust—strong endorsements for bad hires reflect badly on you.

More casual referrals can nudge a resume into view in "Resume Review," but that's about it. I personally don't love making them myself (there isn't a lot of upside), but they aren't harmful if you're clear about your confidence level: "I know George from SprocketSoft; I didn't work with him extensively, but he's very interested in WidgetSoft."

Resumes

Resume formatting is not, in my experience, "make or break" of anything in tech—they can hurt a little, they can help a little, but the content speaks much more than the format. Still, there's no reason not to get them right. Below are the most important points; follow them, edit, tinker, and when you're done, get a peer review, ideally from a senior engineer or manager with interviewing experience. Let your friends, not a hiring manager, catch your mistakes.

Section Order

Sections should be ordered as experience, then skills, then education, because those are the priorities of hiring managers. That observation alone tells you something about the importance of internships for a student: they (usually) weigh more than coursework with hiring managers! If you're early in your career, you may elaborate more on your education (e.g., specific classes and projects); as you mature in the industry, you'll emphasize projects more and schooling less. Lots of people include hobbies; I think they're a nice-to-have and can safely be skipped.

Formatting

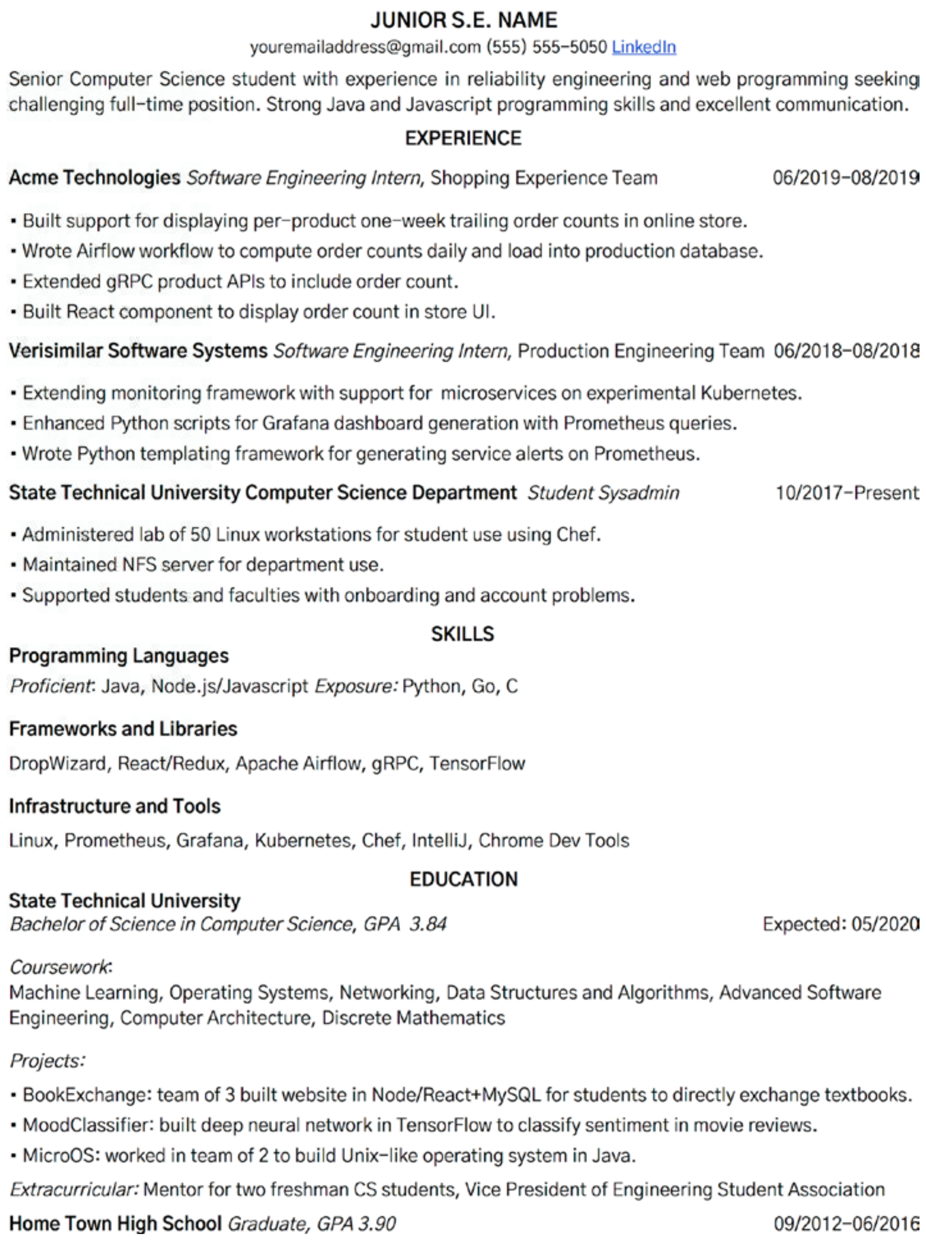
Resumes should be a single page. You can do it. If you are early in your career, you absolutely do not need more than one; the second page just says "I take myself too seriously" (hiring managers really will see it that way). Also, table gridlines give an appearance of amateurism (I can't exactly say why, but they do).

Tell a Story

Emphasize what you delivered, where you led, and the results your projects yielded: managers like signs of autonomy and leadership. Never say "Implemented features and bugfixes," which is well-known to be the most generic line ever added to an engineering resume; help the reader visualize you solving a big problem or taking a project from conception to delivery, not sitting passively at your desk waiting for someone to give you a bite-sized task.

Example

Below is an example of a junior engineer's resume; it's not a work of art, but if you're in doubt, you can copy this format.



JUNIOR S.E. NAME

youremailaddress@gmail.com (555) 555-5050 LinkedIn

Senior Computer Science student with experience in reliability engineering and web programming seeking challenging full-time position. Strong Java and Javascript programming skills and excellent communication.

EXPERIENCE

Acme Technologies *Software Engineering Intern*, Shopping Experience Team 06/2019–08/2019

- Built support for displaying per-product one-week trailing order counts in online store.
- Wrote Airflow workflow to compute order counts daily and load into production database.
- Extended gRPC product APIs to include order count.
- Built React component to display order count in store UI.

Verisimilar Software Systems *Software Engineering Intern*, Production Engineering Team 06/2018–08/2018

- Extending monitoring framework with support for microservices on experimental Kubernetes.
- Enhanced Python scripts for Grafana dashboard generation with Prometheus queries.
- Wrote Python templating framework for generating service alerts on Prometheus.

State Technical University Computer Science Department *Student Sysadmin* 10/2017–Present

- Administered lab of 50 Linux workstations for student use using Chef.
- Maintained NFS server for department use.
- Supported students and faculties with onboarding and account problems.

SKILLS

Programming Languages

Proficient: Java, Node.js/Javascript *Exposure:* Python, Go, C

Frameworks and Libraries

DropWizard, React/Redux, Apache Airflow, gRPC, TensorFlow

Infrastructure and Tools

Linux, Prometheus, Grafana, Kubernetes, Chef, IntelliJ, Chrome Dev Tools

EDUCATION

State Technical University

Bachelor of Science in Computer Science, GPA 3.84 Expected: 05/2020

Coursework:

Machine Learning, Operating Systems, Networking, Data Structures and Algorithms, Advanced Software Engineering, Computer Architecture, Discrete Mathematics

Projects:

- BookExchange: team of 3 built website in Node/React+MySQL for students to directly exchange textbooks.
- MoodClassifier: built deep neural network in TensorFlow to classify sentiment in movie reviews.
- MicroOS: worked in team of 2 to build Unix-like operating system in Java.

Extracurricular: Mentor for two freshman CS students, Vice President of Engineering Student Association

Home Town High School *Graduate, GPA 3.90* 09/2012–06/2016

Passing Engineering Interviews

This section is a brief overview of how to pass software engineering interviews. It will discuss what interviewers look for, what they'll ask, how to prepare, and how to behave during the interview. A later chapter will cover this subject from the interviewer's perspective. Whole books have been written on this subject, and as you look for your first job, you should read one (look at the Appendix to this section).

What They're Looking For

Software engineering interviews usually look for two things: ability and "culture fit." As we'll discuss in "Interviewing Software Engineers," neither is well understood, and neither is sought in a coherent way. However, you don't need to solve that problem for the industry: you need to pass interviews, which you can easily do with preparation.

Hiring managers look for several dimensions of ability. They are, in roughly decreasing order of priority

- Coding/debugging measured by coding on the fly in interviews and sometimes by a take-home coding problem
- Design/architecture measured by a design exercise in an interview
- Communication measured by how clearly you express your ideas in interviews
- Domain knowledge measured by factual questions and design exercises

"Culture fit," often and correctly maligned as a tool of conscious or subconscious discrimination, usually means three things:

- Enthusiasm for the role
- Positive attitude and friendliness
- Whatever interviewers happen to like

All three are measured by questions about your interests and goals and by the interviewers' general sense of your attitude.

The relative weights of domain knowledge, culture fit, and "raw ability" (i.e., coding and debugging) vary considerably by company and interviewer, but by and large, pure interview coding skill, that is, the ability to solve coding problems on the fly while talking about what you're doing in a pleasing way, is priority #1 for junior hires, and as of this writing, many companies are willing to give "smart people" a try at a specialization they haven't practiced before.

Acing Coding Interviews

For passing interviews, coding is king. That is to say, interview coding. Programming interviews are a kind of sanitized, stylized coding, a performance art where you have 30–60 minutes to solve a problem chosen by the interviewer while talking through your work; there's almost always some kind of tricky algorithmic problem at the core of the question.

You should on no account confuse interview coding with the day-to-day work of a software engineer, which is far messier, mostly driven by the behavior of existing code, mostly about integrations and debugging, and almost never about cracking a tricky algorithmic problem, which I personally do just a couple of times per year.

On my bad days, I'm outraged by the lack of realism of coding problems and the way they favor people who are blessed with the ability to be calm under pressure and a gift for oratory, neither being skills that come up on a daily basis when doing the real job. However, interviewers need to ask something, and while these interviews may not be that realistic, they are reasonably easy to prepare for; you should think of a coding interview as a performance art that you can easily excel at with practice.

Preparation

Here's how you prepare for technical interviews, in decreasing priority order; because coding interviews are fairly predictable, most engineers I know, no matter how experienced, prepare roughly the same way:

- Solve a bunch of coding problems, with real code, to get your brain in the groove of time-pressured problem-solving. Sites like leetcode.com have large banks of practice questions; question quality varies, but if you do 50 problems end to end, you'll be more than ready.
- Study your CS fundamentals, especially linked lists, hash tables, trees, sorting, and the (Big-O) analysis of the memory and runtime of all of the above. Brush up on dynamic programming if you're feeling energetic.
- Brush up on the specific domain of the job you're applying for, and prepare to discuss the standard technologies architectures in that space.
- Practice talking through what you're doing to get used to the performance aspect of interviewing; have a friend grill you in a mock interview if you can.