Lessons from real world

And some unsolicited advices

Part 1 : Some real world ~~lessons~~ experience

What did I do?

- I am working at 99.co. I have done, doing-
  - Mobile App Development
  - Backend Development (APIs, System Design)
  - Infrastructure Management
    - Server provisioning, setup, monitoring,
  - Lots of Glue work
    - Hard to quantify but essential
  - Some leadership work for the engineering teams
  - Some mentorship work for junior developers
  - Firefighting
  - Interviewing

Product vs Service (Agency) Companies

- Some differences between them
  - Sense of ownership
  - Exposure to new technologies
  - Exposure to the different type of customers
  - Long term maintenance vs short/mid term maintenance
- Some Product Companies
  - Google, Facebook,
- Some Service(Agencies)
  - Infosys, Accenture etc
- I am going to focus more on the product side.

What happens at a product company?

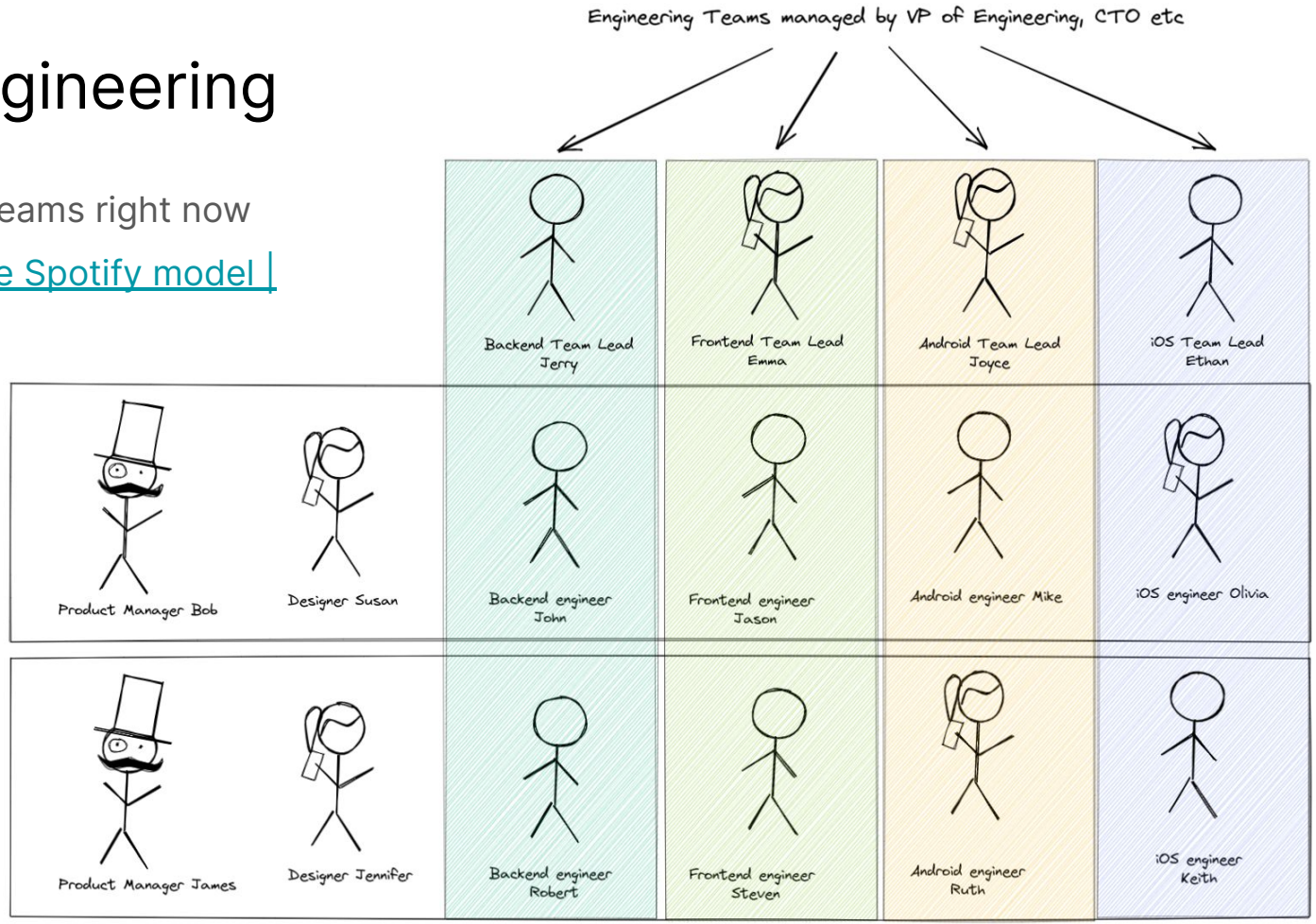
- Several Product Teams working on different parts of the products
  - Some User facing features
  - Some Internal tools
  - Some Growth specific features
- Other non-product teams include
  - Customer Support teams
  - Marketing
  - Sales
  - Finance
  - Operation/HR

What are Product Teams?

- Product Teams are composed of
  - Product Manager (s)
  - Product Designer/UX Researcher
  - Backend Engineers
  - Frontend Engineers
  - iOS Engineers
  - Android Engineers
  - Quality Assurance (QA) Engineers
  - Maybe more !

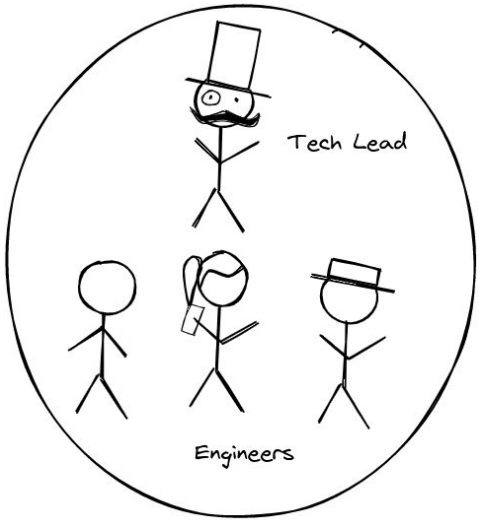
Product x Engineering

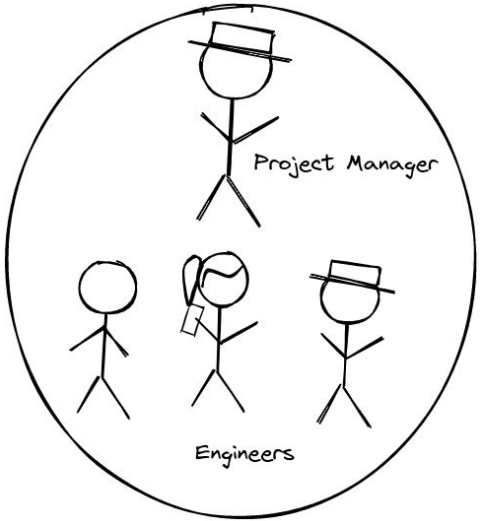
- Several Product teams right now
- Spotify model [The Spotify model | Atlassian]



Product x Engineering

- There are other models

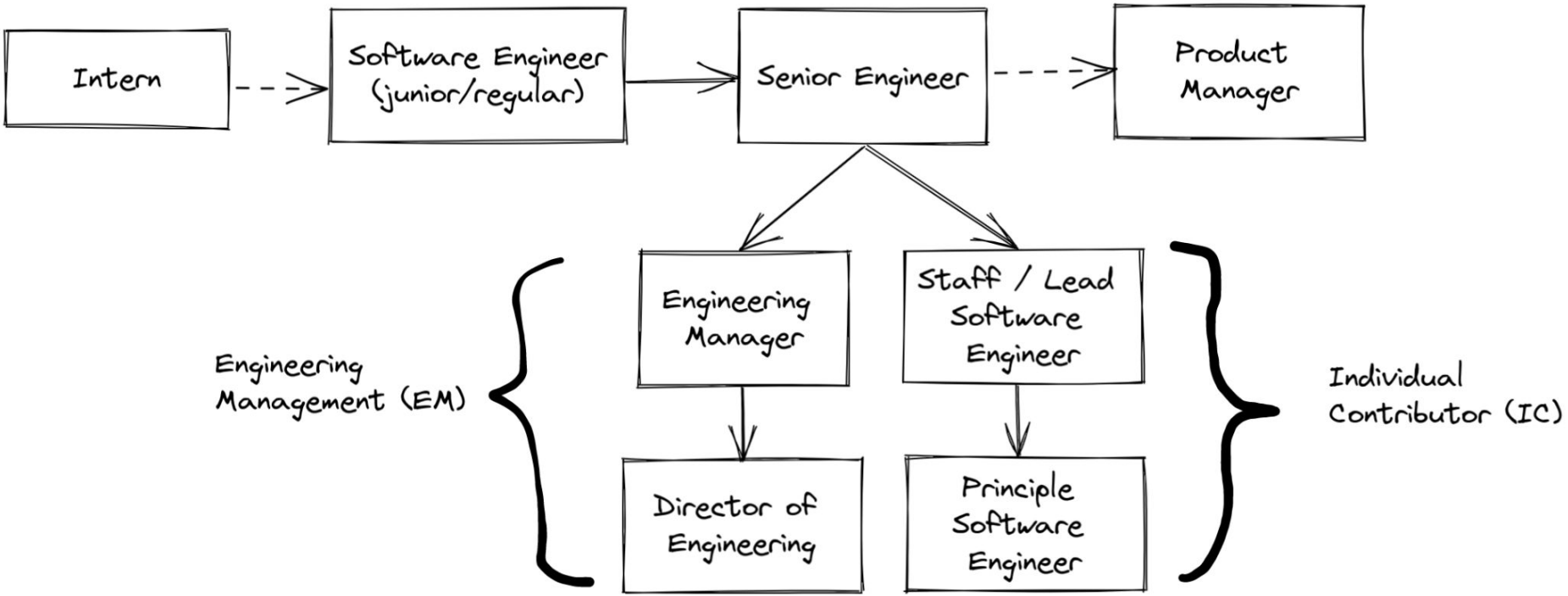




What’s the role of Engineering

- Plan/brainstorm the product roadmap
  - Work together with Product Managers and designers.
- Support the Product Vision
  - Implement the Product features in a correct, reliable way, in a given time.
- Maintain the current Product
  - Make sure things are up and running, bug fix, do routine maintenance of the infrastructure

What are some Engineering career path?



What are some Engineering career path?

- The difference between the Juniors (and Regulars) and Seniors are around -
  - Ownership (of decision, implementation etc)
  - Impact of work
    - Force multiplier
  - Mentorship
  - Mastery
  - Leadership and Management
  - Soft skills

What are some Engineering career path?

- Most engineers wanna only do “technical” stuff. Purely technical, hardcore stuff
  - Writing databases
  - Performance optimization
  - Making things scale for the future 100x growth
  - Writing your own ORM / X Library
- Technical mastery is *very* important but … it is only *half* of the picture.

What’s the other half?

- Communication (Speaking/Writing/Listening/Reading)
- Providing feedback in a constructive way
- Understanding of the product and context
- Being able to reason the trade-offs
- Interviewing
- Onboarding, mentoring other engineers
- Influence. Not authority

But I still only want to do technical stuff ...

- Find the opportunity in where you work
  - Talk to the other engineers. What are their pain points? Slow APIs? Slow Deployment?
  - Find the root causes. Figure out a plan to tackle the problem.
  - Talk to your boss to give you the resources (time, engineering resources etc)
- Plenty of things to improve in a multi-years old product company :-)

Check out these links!

- James Hickey 🇨🇦👨‍💻 on Twitter: "Thread for Junior Developers/Engineers"
- Lorin Hochstein on Twitter: "As a software developer, you may be called upon to perform some of these tasks in your career. How well a CS degree prepares you for these tasks (and whether it even should prepare you for these) is left as an exercise to the reader. 🧵 1/" / Twitter
- An incomplete list of skills senior engineers need, beyond coding

Any Questions so far?

Part 2 : Bits of Unsolicited Advice

1. You will make mistakes

- Not just in Engineering but also in life!
- There is no other way around it
- Find them early.
- Mistakes are *rarely* fatal in our field
  - You can make them, should make them, learn from them, laugh it off and move on.
  - **BUT … DO NOT MAKE THE SAME MISTAKE AGAIN.**
- Try your best to be right. It is okay if you made mistakes.
- On the flip, be patient and be welcoming.

Pros are just amateurs who know how to gracefully recover from their mistakes.

- Kevin Kelly

2. Concentrate on Fundamentals

- How to know what are fundamentals?
- One test is “they” have lasted for a long time.
- Another test is “they” are what all the rest of the field can be derived by using the standard methods in the field.

Some fundamentals every programmers should know

- Basic data structures and their time/space complexity
  - List, Dictionary (Hashtable), Set . Implement at least once.
  - Binary Search, Binary Tree, Trie etc
- Source control like Git
  - Are you just remembering a few Git commands ? Do you know how Git works ?
- Command line
  - Basic command line programs. Some Bash scripts.
- Databases and SQL
  - What are DB indexes?
- Basic Networking
  - TCP/IP

Some Computer Science track for self-taught ...

- Check out Teach Yourself Computer Science

TL;DR:

Study all nine subjects below, in roughly the presented order, using either the suggested textbook or video lecture series, but ideally both. Aim for 100-200 hours of study of each topic, then revisit favorites throughout your career 🚀.

Subject	Why study?	Book	Videos
Programming	Don't be the person who "never quite understood" something like recursion.	*Structure and Interpretation of Computer Programs*	Brian Harvey's Berkeley CS 61A
Computer Architecture	If you don't have a solid mental model of how a computer actually works, all of your higher-level abstractions will be brittle.	*Computer Systems: A Programmer's Perspective*	Berkeley CS 61C
Algorithms and Data Structures	If you don't know how to use ubiquitous data structures like stacks, queues, trees, and graphs, you won't be able to solve challenging problems.	*The Algorithm Design Manual*	Steven Skiena's lectures
Math for CS	CS is basically a runaway branch of applied math, so learning math will give you a competitive advantage.	*Mathematics for Computer Science*	Tom Leighton's MIT 6.042J
Operating Systems	Most of the code you write is run by an operating system, so you should know how those interact.	*Operating Systems: Three Easy Pieces*	Berkeley CS 162
Computer Networking	The Internet turned out to be a big deal: understand how it works to unlock its full potential.	*Computer Networking: A Top-Down Approach*	Stanford CS 144
Databases	Data is at the heart of most significant programs, but few understand how database systems actually work.	*Readings in Database Systems*	Joe Hellerstein's Berkeley CS 186
Languages and Compilers	If you understand how languages and compilers actually work, you'll write better code and learn new languages more easily.	*Crafting Interpreters*	Alex Aiken's course on edX
Distributed Systems	These days, *most* systems are distributed systems.	*Designing Data-Intensive Applications* by Martin Kleppmann	MIT 6.824

How to start actually?

- Read books
- Work on personal projects
  - Words are not enough sometimes. You gotta do it actually.
- If you are stuck, get help from other people who have done before.
- Don’t do it alone. Do it together with like minded people.

3. Skills require practice

- Practice deliberately.
- Do things that *hurt*
  - E.g Learning how to write a compiler is hard and also hurt your brain. It's also very rewarding!
- It takes time.
- Do it, analyze yourself. Re-do it.

Sometimes reading books/articles is *just* not enough

- You have to actually do the things.
- Build apps, projects.
- Over time, you learn how to predict the problems, and, when you can’t, you arrive at solutions with less errors and more assurance.

How to choose what to do?

- So. Many. Things. Out. There.
  - GitHub - karan/Projects: A list of practical projects that anyone can solve in any programming language.
  - GitHub - florinpop17/app-ideas: A Collection of application ideas which can be used to improve your coding skills.
- Priorities – what's the important stuffs?
  - Ask yourself honestly. E.g Things I Don't Know as of 2018 — Overreacted
- You will not have time for everything.
  - Ask yourself your 9-5 job offers you opportunities for improvement.

Be a good software engineer no matter what you do

- Having strong programming fundamentals and debugging skills help you a long way.
  - E.g machine learning engineer/data scientist who don’t know SQL or how to debug the CI issues.
  - Frontend engineer/Backend engineer who don’t know how to deploy the apps yourself.
- Otherwise, your productivity is impaired.

Work on things at the edge of your capabilities

- Work on things that are slightly harder than what you already know how to do.
- If you ever work on something in your comfort zone, you can spend 10 years doing the same thing over and over again, and never really improve.

A few ideas ...

- Compilers 🐉
- Language internals
  - Python Internals
- Library framework internals
  - Tornado (web framework)
  - How does “asynchronous” library works
- What is this “Docker” thing work
- What happens when you type google.com in the browser ??
- What is “HTTP”?

4. Learn in Public

- The fastest way to improve your skill.
- Write what you learned!
- Teach other people
- Give talks.
- Make videos.

Learn in Public

“Try your best to be right, but don't worry when you're wrong. Repeatedly. If you feel uncomfortable, or like an impostor, good. You're pushing yourself. Don't assume you know everything, but try your best anyway, and let the internet correct you when you are inevitably wrong. Wear your noobyness on your sleeve.”

5. Focus on 4 things

A. Connections
    a. Who I know and who knows me
B. Skills
    a. The value I can provide to the business and the world
C. Assets
    a. Money, Equity
D. Portfolio
    a. Things that I have made (articles, podcasts, books, talks, side projects etc)

6. A few other advice

- Work on personal productivity/velocity
  - Typing speed. Most of us spend a lot of time typing. (40 WPM vs 100 WPM)
  - Learn how to use your tools efficiently
  - The Mythical 10x Programmer
- Learn how to write well, especially Technical writing
- Read a lot of long contents and less short contents.

A few Great reads

- How You Know
- How to Think for Yourself
- The days are long but the decades are short by Sam Altman
- You and Your Research
- Investing on Principle by Bret Victor + Video
- What to learn
- Definitions of Educated Person - Calm Hill
- The Illusion of Knowledge - Calm Hill
- Know your values - Calm Hill
- Evolution of Programming Knowledge Level - Calm Hill
- "So Good They Can't Ignore You" Summary - Commonplace

A few Great books

- The Last Lecture
- Dune
- Endurance: Shackleton's Incredible Voyage by Alfred Lansing
- "What Do You Care What Other People Think?": Further Adventures of a Curious Character by Richard P. Feynman

Thank you for your time!