Hassan Habib

A Collection of Articles from 20 Years of Experiences in the Software Industry

Lessons Learned in the Software Industry

Lessons Learned in the Software Industry

Hassan Habib

Free to read online at hassanhabib.com/books/lessons-learned

Thank you, Lord for all you taught me. Forgive me if I forget anything

Contents

1	An Open Letter to Software Engineers
2	Employees & Engineers
3	Software Engineering from Scratch!
4	IF IT WORKS, BREAK IT!
5	Purposeful Engineering
6	Uncle Bob in the Market
7	Cowboys & Mad Scientists
8	Beyond Agile Software Development Methodology
9	The Ping Pong Table
10	Fat Apps!
11	The Fish Mentality!
12	Software Doctors
13	Mr. Sam!
14	How to Effectively Connect to Each Other?
15	Time-Driven Engineers
16	Foreign Code
17	Would You Hire Your Manager?
18	The Lifecycle of Technology
19	The Perfect Software

20	Chickens, Half-Ways, Almost-Theres and Winners
21	Failure is a Mindset
22	Pain & Passion
23	Innovation Engineered
24	Entrepreneurial Innovative Engineering
25	Systematic Innovation
26	Code Rub
27	The Tribe
28	Sandcastles
29	Unbiased Compilers
30	Pure Coding
31	Reusable Modularity
32	Time-Travel-ish

CHAPTER 1

An Open Letter to Software Engineers

Dear Engineers,

Whether you have been in the industry for a day or a decade this letter is for you.

The software industry is one of the toughest industries out there from an intellectual perspective.

Unfortunately, many who begin their computer science degree with high hopes drop out, or even complete the degree only to go a different route because the challenge of the industry is surprisingly overwhelming and demanding. In spite of the fact that the market is so enticing and the demand for engineers is so high, the field is intimidating. With the mix of math/logic-based science and a need for a highly creative mind to be a successful computer science engineer, a high percentage of would-be programmers leave the field. This is so unfortunate because the market is designed in a way that is exponentially expanding everyday as enterprises, small and big businesses, governments and every entity imaginable become more and more digitized and computerized. Everything from our airplanes and cars to our trash cans and vacuum cleaners are now programmable.

Being lucky enough to be a successful engineer is better than being a kid in the candy store - it's like solving puzzles and racing against time every single day.

Deadlines, designs, retrospective meetings, stand-ups, defects, features and tasks and many more terms and methodologies that govern the day of a software engineer and while that may sound daunting to some, to the true software engineer with a passion for programming, this is a dream come true.

You're being paid a tremendous amount of money because you are some of the smartest people on the planet; you are one of the few 0.25% of the population of the planet and the entire population of the planet relies on your work to run their

day to day lives!

I'm writing you this letter to help you understand your worth, your true value, and the huge contribution you make to this world. At the same time, I want to tell you how you should ensure your continuous growth wherever you work and whatever you do in this industry.

There are three important conditions you need to make sure your current position satisfies.

If even one of these conditions isn't satisfied, you are wasting a finite resource, which is your life.

Your Professional Growth

No, I'm not talking about the autocratic junk that people call promotions, neither am I talking about moving to a shiny office or having a fancy new title. None of that matters in our industry. You know what matters? How relevant you currently are and how adaptive you can be to a forever changing and viciously competitive industry.

I've seen people who have been working in this industry for over three decades and their knowledge remains something from the stone age. The very corporations that hired them as young ambitious engineers are also the same companies that left them aged bare-boned useless cogs who have no idea how to keep up with technology. Sadly, the best thing they could do is to find some vague similarities between what they did twenty years ago and what people need today, then ramble their tales, sending everyone in the room yawning as they peruse their phones for more interesting content.

If you don't want to end up like these outdated engineers, or fear you may already be one, keep reading.

Let's talk about how these cogs ended up like this in the first place.

If you find you have been tasked with maintaining an archaic system with no plans in the near future to modernize, or rewrite the system, know for a fact that you're being used as a cog. Your time is being terribly wasted over something that you'll

never need again wherever you go within the same organization, or in any other software company.

If you're tasked with fixing data without tackling the actual problem that's causing that data to be buggy, know that you're going down the road of being just another cog in that system.

You and the data producing code that's producing that data are the same.

In other words, you're a machine, a toaster! You will soon be tossed away for another "toaster" to replace you!

Staying relevant in the software industry requires a lot of self-discipline, and a desire for continuous self-growth. That means reading books, perusing articles, watching videos, attending conferences, practicing coding, and contributing to projects to stay up to date with the changing industry. Yet it also has to be a part of the best eight hours of your day, the most productive hours when your brain is actually ready to do something great, but instead it's tasked with doing something that can be automated.

If you're tasked during the best eight hours of your day with dull, non-challenging repetitive tasks, you're wasting your time, and you have two options:

1. Push for a serious change in the projects you're working on. Ask questions about why, in an industry that's automating everything in our lives, you have to do the same tasks every day. Find out the root cause some dull task keeps landing on your desk (or anyone's desk) and see if it can be eliminated. Ask why those tasks exist and see if there are solutions to eliminate or at least minimize those tasks.

There will always be some tasks that no one likes to do, like a bad code that caused bad data to happen, or a failure that resulted in some damage. It's okay for that to be a part of your day, we've all been there.

But if that's the *only* thing you do, then something is seriously wrong and you need to start pushing for a change now.

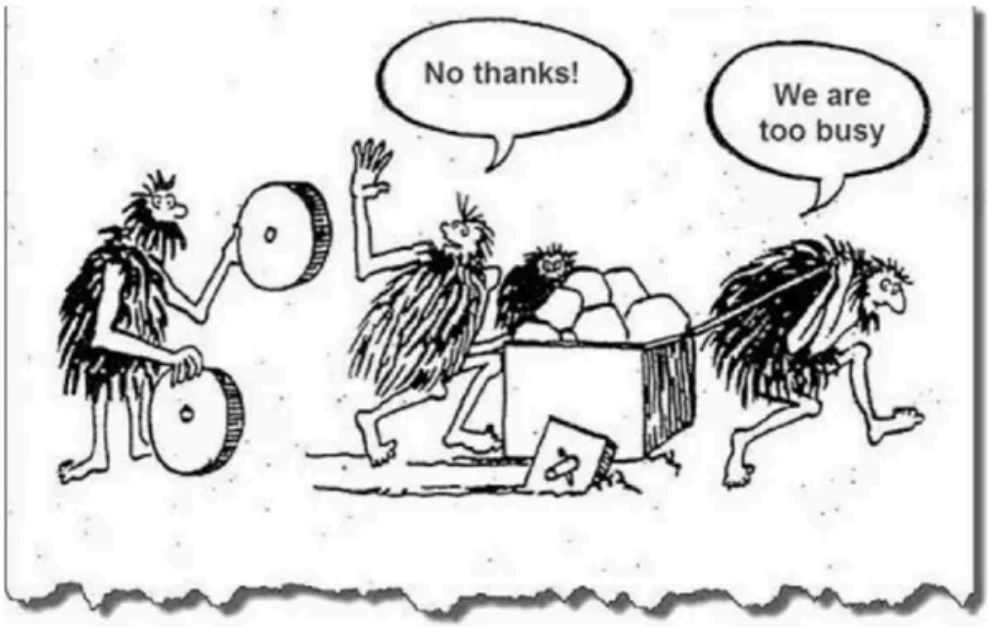
In this case, when you push for change you might run into one of two scenarios:

a. People listening to you and starting to support you and encourage you to find a better solution. If you run into this scenario, you're in a good company, your voice actually matters and you can inspire change, and you should continue the effort and learn as you go about the domain you're working with – but be careful my friend from fake promises and deceiving leadership, these are worse than the ones who shut you down and say no to your face when you try to start a discussion about change and modernization.

They are worse because they'll waste your time, and give you fake promises and deceive you to go pursue your success, they will have you continue to do some random cog-like work whilst you're thinking there's some light at the end of the road … there isn't!

b. The second scenario is when people simply say, no we can't afford change, we're "too busy doing the real work" or "let's just keep the business running" – these people are a good example for you to know immediately that every minute you spend with them is a complete and utter waste of time and I can't tell you how much it irreparably damages your career.

Did I scare you? I'm sorry. Here's a funny picture for you:



2. The other option you have if you don't want to push for a change is to start exploring a different place, but this time you have to learn to ask your interviewer the right questions.

Talking about how to find out whether your interviewer is trying to sell you a "coggy" job or not might need another article, but for now, just follow your gut and be vigilant when you pursue another career, make sure you don't jump ship too much, it gives even good employers an impression that you are unwilling to stay long enough, which will make them question whether it's worth the investment to hire you in the first place.

Every engineer needs to understand that software engineering isn't exactly an eight to five job and your brain doesn't really stop thinking about the code you were writing, or the design problems you were solving during the day when you go back home.

Engineering is actually a lot deeper than that. The software problems become your problems and you take not being able to solve them very personally. You cheer up when you fix a bug or implement a new feature, these are all great things to have as an engineer.

Because engineering is that deep into our lives, we need to ensure we nurture it, sharpen our skills, continue to learn new ones, engage in discussions with friends and colleagues about the best solution to the biggest problems.

It's your responsibility and the responsibility of those with whom you work to allow these discussions to happen in order for growth to be accomplished.

If I were to summarize that first condition for you, I'd say; *don't let anyone waste your time or undermine your potential.* The very fact that you've made it that far and actually landed a job as an engineer makes you one of the tough 0.2% of people in this entire planet that could actually make it that far – so you're an absolute superhero. Don't let anyone make you think otherwise!

Your Value

The second most important condition is whether you're being valued on your team and by the company for which you work.

Don't be just a number, and don't let anyone undermine your opinion, or make you think your ideas on any certain subject don't matter. Demand respect and in turn, be very respectful to others as respect reciprocates.

Make sure your voice is heard by as many people as possible among your team and within your organization. Do not let anyone silence you, or phase you out of the picture.

If the place at which you work doesn't respect you, listen to you, or value your opinion by at least giving it a chance to be examined, then you have to do two things:

1. Bring this issue up in meetings, ask questions about why your opinion wasn't considered, or at least discussed. Make sure you research and proof-of-concept your ideas before representing them.

Provide data. Data always wins for those who follow practical ways of making better solutions. Also, have evidence - show previous experiments with other places that used the same solution, or similar to the same problem.

Be diligent and make sure everyone hears your concerns about not being heard, valued, or respected. Take care never to put yourself in a place where you'll be undermined.

2. After doing the first step, watch, wait, and observe change. See how your organization and management/leadership reacts – if they care they'll change, if they don't care then neither should you and you're better off going somewhere where every employee's opinion is valued and respected.

Don't tell yourself to wait for just a little while until something better comes, or there may be something better around the corner. The psychological toll on you from being marginalized will hurt your self-respect and confidence for a long time, and maybe irreparably – so be careful there. The downward spiral can be a dangerous, fast, slippery slope.

If you're respected in your organization, but you see others not being offered the same respect, say something. Maybe one day it's going to be your turn, so make sure everyone gets treated fairly and all voices are heard, respected, and given fair opportunities.

Your Well-Being

You should feel cared for in every way - in terms of compensation, health, education, and such. Do you work in a good place? Are you comfortable in your environment? Are you offered better ways to commute?

In short, does your organization actually care about your well-being and satisfaction.

In this industry, any distractions might hurt your productivity. It's the responsibility of your employer to make sure you have as little distractions as possible, in life, at work, or any other way as much as they can.

As a software engineer here are just a few of the things you should expect from your employer:

1. You're being paid fairly, based on your position and responsibilities, but you should also check out the market and consult with recruiters in the area and ask them about the average salary for your position. Demand a salary that's fair and equivalent to your skillset and the market. Recruiters are your best bet when it comes to understanding the market, so make sure you stay close and communicate with them - they are always helpful.

2. Your workplace should be comfortable. Don't ever settle for poor chairs or small desks. Your work space should fit you ergonomically so your body does not experience pain or discomfort. Your job is very important and requires a ton of focus, and that requires comfort. If you're not comfortable you will not be able to perform your job the way you should, so demand properly fitting chairs, correctly adjusted monitors, keyboards, desktops, footrests if necessary, and any other accessories you may need. Your workstation should be set to your comfort on your first day of work - expect nothing less.

3. Are you being taken care of from a health and education perspective? Does your work take care of your health issues? Does it allow you to have a proper vacation time for yourself, or when you have a baby or needing sometime to mourn a loved one? In other words, do they care about your mental health and well-being as a human? All of these things play a big role of whether you should

continue to be where you work. You should never have to fight to make it happen - find a place where they respect and value your skills enough to offer perks that show respect to you in every way.

Software engineers are the money-makers in the software industry, everything and everyone else revolves around what engineers create. Marketers, designers, managers, administrators, vice presidents, and CEO, none of that matters if there's no software. And guess who makes the software? You! Therefore, you should be treated with utmost respect and provided the best comfort levels your employer can afford to offer. Never forget that. You are a very, very important component of your company, ergo, that should reflect on the tools and amenities you are provided in order to do your job well.

At the end of the day, remember that your time is very limited, and your life and genius is best spent on something that matters and with those who value your talent, and invest in your growth and look forward to what you can be.

Don't sell yourself short. Don't be fooled. Stay vigilant and informed.

Be kind to everyone, but remember there's a fine line between being observed as kind and weak. Make sure you make that abundantly clear and don't think about ways to be the best in your team, but rather think how to be the best for your team.

After all, it's those who work with you every day who are the ones that know everything about what you can do, and decide to continue to work and grow with you!

It's their implicit way of saying, we like to work with you. Therefore, make sure you communicate with your team, voice your concerns, create a public opinion, and engage in positive discussions to change everything for the good!

Remember that when you try to push for change, you'll be faced by those who try to silence you, intimidate you or shut you down or accuse you for not being grateful or thankful for what you got, Even though you work for their own benefit not just yours. But if you're patient, fearless and kind good things will happen, because in every successful organization out there, there will be those who feel what you feel, and aspire to what you aspire to, so all it takes is just for you to shout a little louder, so they can hear you and help you get to where you want to go.

Sincerely yours,

Hassan Habib

CHAPTER 2

Employees & Engineers

In every profession in our modern world, there are two distinguishable types of individuals that surface to the top and drive any industry into one direction or another.

Few weeks ago, I had an interesting discussion with a friend of mine who happens to be a lawyer – we talked about several things and exchanged experiences in both the software and the legal industries, then I naturally brought up the topic of mediocrity – and whether it has sunk it's teeth into the legal industry as much as it did in so many others.

My friend said: Every. Single. Day.

As much as I've known my friend to be an honest hard working lawyer, standing up for what is right and has a non-negotiable integrity, I had to ask the question: did you have to cut a corner at some point?

My friend said: integrity is not about doing the right thing when you can. Neither it is about doing it when someone is looking, integrity is going the extra mile without expecting any form of reward.

Looking throughout history, I have realized that there have been some instances where people got punished, vilified, and even exiled for the doing the right thing.

But the kind of people that would go the extra mile despite all the blockers, are the same kind I call Engineers in every industry.

They are not engineers in the traditional sense of the word, they are engineers as I define them to be the engines of success in every industry everywhere.

Because these supreme engines will keep pushing for as long as they have power and energy to push, until the last drop of force is gone and there are no other options but to halt.

Halting might be an option for a machinery engine, but not for an engineer driven by the force of passion to push for the impossible, and change the course of human history, once more.

On the other hand, there is this other kind that follows orders without questioning, work just hard enough to get a temporary benefit – these are the very same ones that would try to stop the engineers from progressing, because it goes against their own personal benefit. It makes them look bad in front of their *masters* to have someone else making a better progress, going further and faster, because they don't have the energy for it, it requires passion and true belief in a goal that's beyond the horizon.

This other type is the type I call Employees.

Dictionaries define an employee as a person working for another person or a business firm for pay.

The very definition of the word makes the sole purpose of someone working for a person or a business is to be the monetary benefit. And therefore, engineers can never be employees.

Because engineers have much higher purpose, and more noble goals than the monetary benefits – the monetary benefits are a side-effect, not really the end goal.

Because no matter how much money you make, and how high of a position you have, and how popular you may be, if you are not working towards innovating new ways to improve and maintain the ever threatened human kind and it's progress, then eventually your money will go away to your inheritors, and your position will be filled after your retirement, and your popularity will start to decline in favor of the new shining star in the arena.

In fact, I have seen throughout my life, people who wouldn't even bother to look at their bank accounts, don't care how much they were being paid and couldn't care less about their "benefits" – all they cared about is to be a part of a larger team to create a larger impact.

It's the purpose that drives these engineers to become a part of history, to be remembered as the visionary vanguards, who never gave up, never settled for less and worked hard to achieve the impossible.

Someone might ask: how do you distinguish an engineer from an employee?

Here is my two cents ...

Throughout two decades I have spent in the software industry, I have noticed few *symptoms* that may indicate someone has a case of being an employee, it made them very easy to spot compared to an engineer let's have a look.

Impact Chasers

The engineers that I'm proud to have known, worked with, and proud to be in the same industry as they are, are impact-chasers – they don't let a day go by without maximizing their impact in so many different directions.

They scale by helping other engineers understand the technology, the tools, and the standards to build higher quality software.

They increase their impact by exploring newer ideas, untapped markets, unknown regions of the world where improvements can be made, and happiness of the individuals can be maximized.

A true engineer's goal is to engineer a world where humanity can survive and continue to evolve – it less about themselves, and more about those around them.

An engineer's ultimate goal to have an impact on individuals they might never meet but would still benefit from their creation.

An employee on the other hand, is a clock-watcher, selfish, lonely creature, feeding off of mediocrity and sycophantism – they chase nothing but their own self-benefit, care less about impact and care more about what their *masters* think of them.

If they are well-paid, promoted and rewarded, they will do their very best to maintain their positions, but very little do they actually push for a real impact, a serious improvement or a true meaning to their very existence.

And if you ask them, they'd find a thousand ways to legitimize their position, looking around gathering excuses for their incompetence – they would call you crazy, unhealthy or even evil for seeking more, wanting more and willing to do more for everyone.

When you see someone caring more about the time they spend on a task, more than completing the task and making sure the goals are actually achieved, know that they are employees, clock-watchers performing tasks like a cog in a machine without passion or soul or purpose.

An employee will work just hard enough to get rewarded, promoted or recognized, but they couldn't care less about an actual impact, an achievement, or a true value they would provide.

Engineering is a way of Life

Engineers live the values they believe in. engineering is a way of life to them, it impacts their relationships, work environments and everything they touch.

You will see intelligence in everything they do and everything they have, it's not just a job to them.

These engineers are the ones you see pushing open-source projects to create a value and an impact, some of them go the extra mile to hide their identities as they couldn't care less about even a recognition from someone about the amazing work they do to change the world for the better.

They are the very same engineers you will see volunteering to help a small business with a website, or a poor man with his sales or a student with their studies.

An engineer doesn't really stop being an engineer once they are done with their day job, the continue to improve, innovate, extend and build better and stronger bridges with their communities and families to gain a larger impact, or inspire someone to succeed, change their lives, maximize someone's happiness or spread their knowledge and experience for the rest of the world.

An engineer is the engine for inspiration and motivation to those around him, regardless of what they are doing and how they are doing it, they invest in everyone's win, not just themselves.

An employee's day ends at the moment they leave their workplace. They invest less in their skillsets, and even less in the lives of those around them.

In an employee's day, there are always those whom they *have* to work or live with, they make lesser and lesser choices about their own lives like a lifeless, helpless tree branch, pushed around in a raging ocean without a destination – cut away from the tree of the life, lonely despite floating with other cut branches and weak despite it's size.

Employees are often stuck in a pointless, endless, and self-defeating pursuit – a rat race without a true meaning or value.

Self-improvement, self-growth then expansion and scale is some of the top priorities of an engineer, they don't let a day go by without pushing the needle forward, learning and teaching, understanding and doing – loving and impacting wherever they go, and to whoever to come in contact with.

CHAPTER 3

Software Engineering from Scratch!

For those of you who love computers, who love programming, I would like to share something that is vital for you to know if you would like to make these passions your career. I can help you learn exactly what you need to do to become a successful software engineer. I will give you five crucial steps that will allow you to be well on your way to a successful and lucrative programming career.

Where to start? How do you land that excellent job opportunity? First of all, the most critical piece in these five steps is obtaining your bachelor's degree computer science and technology. Yes, as this field expands, a master's degree or a Ph.D. is even better, but of course we have to begin with the bachelor's degree. While this is often viewed as the most difficult, I would like to introduce you to an awesome resource that will make this possible from anywhere in the world and for almost no financial cost! Additionally, because software development was born in the west – specifically the United States – having a degree from there is also nice to have on a resume. Whether you are in the United States or not, a fantastic resource to get a bachelor's and/or master's degree is from The University of the People (**http://www.uopeople.edu**). Anyone from anywhere can apply to this *accredited* university – and believe it or not – it is free. There is a $100 exam fee, but all of your learning can be done online at no cost. That is the first step.

Next, as a way to build your career, the degree will get you started, but this is by no means the best jumping off point of a successful career. Many students think once the degree is in hand, the job offers will simply flow in, but do not sell yourself short in this way. The degree can open the doors for you, but it is definitely not the final step. The degree can give you a legitimate certificate that exemplifies what you know and you can easily discuss things like algorithms and asymptotic time with other professionals, but you still need to expand your credentials. The second step is to obtain certification from the corporations that provide such learning tools.

These help you to be recognized as a valued professional. While I personally am a .NET developer and this is where my particular interests are, I will not be talking about PHP or Java, even though Oracle does provide Java certificates and W3C schools provide PHP certificates. Microsoft, however, offers a virtual academy **www.microsoftvirtualacademy.com** that can kick-start to your learning. Each video guides your knowledge toward a specific exam. When you become Microsoft certified that increases the chance that you can land great jobs – even with Microsoft itself, or with Apple, Amazon, Google, and other highly recognized business entities.

Thus far, you have two steps toward an awesome career – one practical and one theoretical. The third step is to read. It is vital to keep expanding your knowledge base in order to keep up with the fast paced and quick changing dynamics of software development. Without this step, you risk becoming an ignorant programmer – and believe me – there are plenty of ignorant programmers out there. They just cannot or will not keep up and they continue to use outdated methods in their work. For example, they get stuck using 4-loops rather than using a Lambda or LINQ expressions Even if you do not understand these examples, you will in the future by following these steps.

So you have the degree, certifications from Microsoft Virtual Academy, and you are reading to keep up. If you are concerned about the cost of technical reading materials, you can check out **www.freecomputerbooks.com** or even find inexpensive books on Amazon. Understand that you do have to invest in your mind and the payoff will be phenomenal. Know too, that while you may not be making money right away, by investing in yourself for four or five years, you will soon be on your way to a more lucrative lifestyle. The beautiful thing about programming is that you do not have to begin with much. You need a computer and a keyboard and you are good to go! So up front, if you really want to make an impact in the technology world, you have to invest in your mind. All the technology we use today began as an idea. Windows began as an idea. Google began as an idea. Facebook, WhatsApp, and such, all began as an idea so by continuing to read and learn, you open your mind up to the possibilities that begin as your idea and allow you to contribute to the technology we will be using in the future.

The fourth step is to make sure you have the tools necessary to gain practical experience. The best way to access the tools you need is the free software available to students. MSDN is a great subscription service and Microsoft provides free software through DreamSpark (**www.dreamspark.com**) if you are enrolled in a university. Using your student ID from The University of the People (which is a Microsoft partner) you can get Visual Studio or Microsoft SQL Server or any other software you need for free.

The last step is for you to get involved in open source projects. You will learn from experience what you cannot learn from books, college, reading, or even from your certifications. Get involved by going to **www.github.com** and learn version control by downloading C# projects and contributing the genius you have to offer to the world. A little warning here, you may be criticized because people can be sharp and hurtful in their feedback, but keep in mind that this too can be used to enhance your learning. No one there is getting paid, there are no politics, no working hours, no bosses, so people tend to be frank and up front. Just keep in mind that your sole purpose there is to learn, not to make friends or secure a job – although some of the greatest companies like Amazon tend to look at your GitHub account rather than your resume to see how you actually solve problems and work through issues. If your code is very good, you will get to know some great people who don't necessarily know how to express their knowledge through a resume, but through the proof of their excellent coding skills.

So by committing to these five steps for four or five years, your life can change dramatically. You need not view programming as proverbial "rocket science" – you just need to work the steps and you will be well on your way to an amazing career.

CHAPTER 4

IF IT WORKS, BREAK IT!

(IF IT WORKS BREAK IT!)

Software isn't just the code engineers write, neither that pretty UI users enjoy.

Software is a living organism and a manifestation of a collective cohesive minds that try to create something bigger than themselves to achieve bigger goals.

When we talk about developing software, we need to consider the human aspect of it, the team that invests their time and effort in it and how to keep the business, the engineers and the engineering process all in sync.

Experimentation is a fundamental part of the software development process, it's simply the continuous process of taking an existing business aspect of the software and going out to explore how to could be done better with more relevant technologies and newer mindset.

Experimentation is simply our way of being proactively seeking a better way of doing things which will eventually protect us from disastrous unexpected issues with our system instead of being re-actively trying to remedy rottenness that comes from rarely visited nearly abandoned archaic code that plays a fundamental important part of an existing system.

Take security for instance, would you rather keep up to date with what can possibly cause a security breach (Murphy's Law) or would you rather wait until your system is completely compromised and losing your user base?

Let's take an example of us as humans, when we go through a bad experience in life, we don't go die and bring a new version of ourselves.

We reinvent ourselves, we don't give up, we reflect, and we rise again, we evolve! It's what makes us great!

Software should follow the same pattern, in fact, the continuous process of experimentation and merge into production is a lot cheaper than a system rewrite.

Rewriting systems is a very old-school mindset, it's almost a waterfall-y even if it disguised itself in an agile process.

It's waterfall-y because it means we had to wait until the system is completely useless before we had to start from scratch, instead of trying to continuously replace parts of the system with newer more relevant, efficient code.

It's one of the advantages of software, it can be modular, componentized and flexible.

Sometimes people say, we have a legacy system that we are planning to sunset in the next year or two, we don't want to invest anymore or experiment with new technologies in that system.

That's the wrong way to go, simply because it means you're telling your team that everything they are currently working on is going to be thrown away which brings the team morale down under.

It means that you've failed to keep the system relevant and up to date therefore you're going to throw it all away and start over, and over instead of replacing the older parts with newer parts which keeps the business domain intact instead of having to reinvent the wheel by rethinking everything over again.

It also means that you're telling your team that by rewriting the system with the newest technologies you simply don't need their skill sets anymore, you need new people with new skill sets, the skill sets that you should have taught your existing team through experimentation so you can leverage their existing domain knowledge instead of having to start over again with a completely new team.

This is what makes new architecture patterns such as Microservices and Serverless so appealing and attractive, because it allows that gradual reinvention of the system in a low-risk safe and productive manner.

There will always be a bit of pain trying to keep a new service relevant and compatible with an older legacy service, but if that legacy service is only a year or so old, it wouldn't be as painful as a four- or five-years old legacy service.

Legacy mindsets will produce legacy people that'll be eventually disposed and kicked to the curb turning them into only bug and data fixers maintaining archaic systems with their lives, instead of being exploratory researchers, story-tellers that know how to stay relevant and carry their weight in a forever-changing industry to keep up with the competition.

Here's some of the advantages of experimentation or R&D:

Experimentation for the business:

- Keeping the business relevant and up for competition, easy to push new features and sunset older ones and fix issues.
- Makes the business appealing to the best talented engineers to come and contribute to it, very high engineers retention rates.

Experimentation for the engineers:

- Growth of skill sets, more stories to tell about various technologies.
- Willingness to stay around longer, to learn more and contribute more and being more proactive about technology.

How some legacy systems end up in that bad state?

- **Legacy Mindsets** will produce legacy systems, when people don't try to stay relevant their software will be nothing but a reflection of their mindsets.
- **Incompetence:** I have come across people that don't really want to learn something new, they don't want to spend the effort in staying relevant, therefore they are happy with doing the same thing on the same system for the next twenty years.
- **Blind Spots:** when the business is in control of how (not what) engineers develop the software, it becomes harder to explain why destroying something and rewriting it to produce the same exact existing results is a very important part of maintaining the business. I always say the further you are from touching the code base the less say you should have in how to write it.

How to practice experimentation in a software development process?

There are many ways of practicing experimentation in an agile environment, I'm going to draw a framework here and some examples of how to execute but for each team the ultimate freedom of how to implement this.

Here's some ground rules:

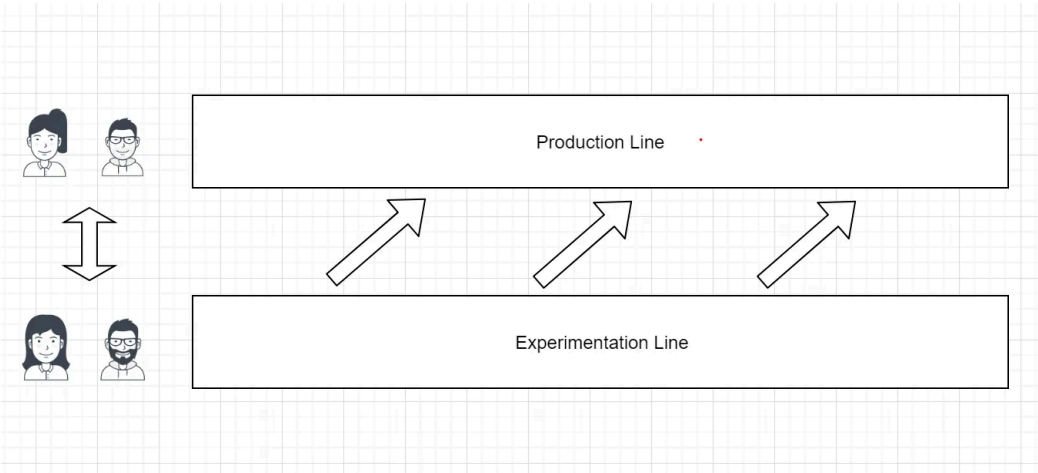
1. In every sprint there should be allocated time and effort to experiment with a new technology, method or architecture or even something as little as a new software pattern to think about doing an existing feature or add a new one in a better more innovative way. The industry never lacks new methods, technologies and ideas.

2. Only few members of the team should be working on the experimentation, so the rest of the team is able to continue working on the current features, defects for the existing software.

3. At the end of every sprint experimentation team should demo their work to the rest of the team and get feedback and improve on their next iteration of experimentation.

4.In next sprint, engineers who did the experimentation should switch to working on the production line, so the others could do their fair share of experimentation as well. This way the entire team gets to try different technologies, ideas, cultivate knowledge and leverage the skill sets of each other and improve the system and make it more appealing and competitive than ever.

Here's a little illustration of the framework:



Zuckerberg once said: “Move fast and break things. Unless you are breaking stuff, you are not moving fast enough.”

This industry is an innovation-based industry, ignited and maintained by innovative minds within innovative cultures that encourages experimentation and research.

In this industry there’s no place for snoozing, incompetence other than cleaning up after even more incompetent individuals who probably don’t even work in the industry anymore, do you want to be that guy? No? then experiment, make mistakes, fail fast and fail cheap, take risks and keep learning!

If you’re environment doesn’t nurture experimentation and innovation then push for it, change the world for the good!

Nadella says: “You need new ideas and you need new capabilities, but the only way you’re going to get those new ideas and new capabilities is if you have a culture that allows you to grow those”

So if this culture doesn’t exist wherever you are, bring it to your place, don’t be a quitter, don’t give up too easy, take risks and engage in changing your workplace, you get a lot more experience and credits for evolving a non-innovative environment into one that nurture it, rather than joining one that’s already there!

CHAPTER 5

Purposeful Engineering

Having software engineering experience without a true purpose behind it is like owning the smartest car, but without a destination.

You keep going and going until you run out of energy, but still end up getting nowhere.

Software engineering today impacts the lives of billions of people around the world – it's tremendous powers can change an entire country into one direction or another; it can protect the planet from eminent dangers and save the lives of millions, as well as being a tool for crime and wrongdoings.

Software engineering is a vehicle for transmitting, processing, distributing, and manipulating data and turning it into information that can heavily influence and oftentimes drive decisions.

Decisions is what we do on daily basis, from the smallest child making a decision about which toy they want to play with, to the highest rank in a government system, making the decision about the future of a country and all the people living in it.

Given the overall history of civilization – technology and therefore software engineering is relatively new to the world, there's only a small percentage of people today that can understand the true power that lies behind being a software engineer – someone who can develop a website or an app today is someone who has the power to make whatever they want visible and available for billions of people around the world at a click of a button.

With that kind of power a lot of the leaders in the software industry have called for responsible engineering, some even proposed having software engineers swear an oath to be responsible about what they make and why they make it with or without pay.

This situation becomes even more complicated at the new age of A.I and Machine learning – soon, every industry in the world shall become automated no matter how complex it is, and the only industry that is going to be left for humans to drive will be software engineering – until we hit the point of singularity.

A responsible software engineering is more broad than purposeful engineering – responsible engineering is simply about being a good cog in a machine, to simply not to do anything harmful and to be more careful not to program defective software that may cost irreparable damages.

Purposeful engineering is more focused on harnessing the power of software engineering and it's reach to ensure the survival of the human kind and it's evolution.

We as human kind today face so many different challenges that threaten our survival – a simple meteor could wipe out the entire planet at a glimpse, and other than the ongoing attempts by visionaries like Musk and others to turn humans into an interplanetary species we don't really have any solution for that very possible danger.

A simple outbreak of a virus like COVID-19 has successfully killed around two million people and infected over fifty-five million others in very few months.

Our labs today and the software that supports the research and experimentation is not intelligent enough to detect examine and produce a cure for a killer virus like COVID-19 in short enough period of time before millions of people have already lost their lives or walking around spreading it further and further.

The prioritization of the overall engineering industry is for the most part going in the wrong direction, there are more engineers today in the finance and stock exchange industries than the research and medical industries.

There are more engineers working on trivial secondary products than the experimentation for inhabiting other planets and having a "Plan B" to avoid an eminent danger to our planet.

Purposeful engineering is a focused form of engineering where all the engineering efforts are targeting the survival of the humankind and its evolution.

Our evolution depends on our adaptability to the forever changing universe around us, not by how much military artilleries we have, neither by how much money we earn. because none of that would matter if a planet-wide disaster occurred.

Our earnings, and our fancy homes, cars or jewelry are not much of use if they were to be completely wiped out from the face of the earth.

Its only 100 seconds till midnight – we are at a very urgent state to focus our powers and our efforts to push towards what matters the most to our survival – our powers, energies and efforts should be mainly focused towards the fields and the industries that turns the clock back and brings us back to a stable state where we can focus on our evolution and growth as one kind.

These fields of focus are the fields that feed these following realms:

Education

Any species that lack intelligence to power up their adaptability to a forever changing environment and threats will eventually cease to exist.

Educational institutions are the factory that produces intelligent individuals that can carry the torch forward and innovate newer ways to ensure our survival and evolution.

This can range from schools and universities to libraries and tools that makes information accessible to the widest range of people in all languages to ensure humanity has the information it needs to create enough experts to expedite the process of resolving the most challenging issues we have today.

I started with education because education is the fuel to all the fields that follow, all the departments and categories of science and research are powered by educated individuals who invested the time and effort to educate themselves and earn the proper degrees to become where they are, where the fate of humanity lies in their hands and what they do in their day-to-day jobs.

Health

While education is one of the highest priorities, the medical department in our world still suffers from so many challenges – as I mentioned earlier, the tools and equipment and the speed of research and innovation in the medical field is behind the times.

Additionally the time it takes a person to do something worth it's while in the medical field is over a decade, it puts the degree seekers in an economical stress with all the heavy costs of getting a medical degree and the time that is spent on traditional century-old ways of teaching the human anatomy, incremental training and adding responsibility takes longer time, additionally a medical student is required to see as many permutations of a medical problem as possible so they have a chance to work it safely on their own when the time comes.

A lot of these issues can be resolved with machine learning and artificial intelligence, it should take one universal system less than few years to digest and *understand* all the permutations of a human condition to drive a better decisions to a successful surgery.

It's the responsibility for software engineers to prioritize the medical research to expedite the process of adaptability to newer challenges and more dangerous viruses threatening the survival of our humankind and our planet.

But more importantly automate surgeries that have, is and will be performed in very similar ways to give a chance to those who do not have access to doctors to live, especially in the most remote areas where it's harder to get the help needed in time.

But the health department covers all things that impacts our health, psychological, physical, and mental.

For instance, protecting our environment and ensuring the well-being of our planet falls right under this umbrella – along with all the efforts some engineers spend into making their software more accessible, more reliable and easier to use for those who suffer from disabilities.

Space

We are at a place in the history of humanity where thinking about transforming into inter-planetary species is not just a dream or a fantasy anymore.

Due to the continuous state of the world from an environmental standpoint, it is really important for all of the engineers to make it their purpose to support the cause of finding a new home for all of us – that's our plan to have the option to survive in case a catastrophic event occurs.

Today, we all live on one planet, one home and under the same conditions and circumstances that impacts all of us at once, like a ravaging virus or an environmental disaster like hurricanes and tsunamis.

If we are thinking together as one nation and do our best to inhabit other planets and give ourselves, children and many, many generations to come to start over and spread across the universe to hopefully increase our chances of survival and growth.

Working towards becoming inter-planetary species shouldn't impact our interest into saving our planet, and investing in producing a better healthcare system for the world – simply because if we can't develop a better system to protect ourselves from killing diseases or maintain the planet we already live in, there's no guarantee that we wouldn't make the same mistakes for any other planet we colonize.

Throughout my life, I have always worked towards connecting with others, sympathize with their issues and make their happiness my ultimate goal.

I have then found that filling people's bellies with food, minds with intelligence and hearts with hope and inspiration is the key to their happiness – it's the key to create an entire generation of people working towards each other's happiness and finding joy in spreading that around the world.

Focusing on Education, Health and Space does just that – on a humankind level, filling peoples minds with intelligences through education, and bellies with food through healthier and safer environment and their hearts with a better tomorrow by looking at the stars through space does wonders.

There are so many different occupations and departments that just falls under these three departments, like farming for creating healthier food, and creating a better environment through environmental studies and all that follows and falls under that ultimate goal – an intelligent, healthy happy species ready for survival and growth for millions of years to come.

Software engineering today is in the heart of every profession and every aspect of our lives, and we, as software engineers need to identify our purposes to create a true impact in our world and engineer our lives, careers and efforts towards these higher purposes.

If you want to measure your worth and impact in this world, think of it in terms of apples.

Some people consume an apple and produce and apple, they come and go, and they don't change anything or anyone's life – these are the ones I call zero-position folks, the go no where and they cause no change.

Some others consume an apple and produce nothing, these are more of a burden than anything else – keep clear from those who are able to be productive, but choose to lay on their backs and watch others work for them, produce for them while they do nothing.

But then comes those who consume and apple an create a farm full of hundreds of thousands of apples, feeding hundreds of thousands of people, filling their hearts with happiness, their bellies with food and their minds with intelligence.

Which one would you like to be? The choice is yours!

CHAPTER 6

Uncle Bob in the Market

Test Driven Development/Design (TDD) has long been established as one of the best software development methodologies. This methodology consists of writing unit tests prior to writing the actual function, then writing code to ensure each test passes. This inevitably indicates the completion of a given feature. Even though this methodology is quite effective and is considered one of the best of programming practices, it remains a narrow vision with respect to the big picture of each product and of the market/user as a whole. In modern software development teams, a typical scenario would be like this - the product owner provides a requirement with very specific acceptance criteria. The developers take on these requirements and begin by writing unit tests that verify that each feature is successful with all of its possible cases. Next, they work on writing the code that will make these tests pass (meaning the feature is complete). They will then go on to refactor the code to make it as readable as possible, so it is easy to read and maintain by other developers. The feature then goes out to the end user and often the user will make additions or changes, forcing the cycle to begin again from square one. Today this case is handled by having the developer saying, "it's okay, we're agile, we will take the changes, write a new user story, scrap all that we have done, and begin again!"

Look at it this way, if you were to build a house, you would begin with the overall design, having your client provide feedback and adjusting as you go, prior to installing security cameras (unit tests) and buying furniture (refactoring).

So it follows that instead of adding unit tests and refactoring before getting the actual feedback from the end user (not the product owner), why not push the testing and refactoring until after the acceptance is received. Not only does this save time and money, it keeps both the developer and the user in sync with what needs to be accomplished.

Some may say, “Well, the product owner has to make sure that the requirements are exactly what the user needs!”

I would say, The end user doesn’t even know what they need until they see something concrete and they will then start forming an educated opinion about what they REALLY want. In other words, even if your product owner was a psychic, he still won’t get what the end-user will really pay for, simply because that this very end user doesn’t really know what to expect and what he wants. It would be a huge waste of time and money to invest in what we, the engineers, think about how the product should be in our perfectly engineered worlds. Instead of actually waiting to get a firm final answer from the end-user, (y’know the dude who’s going to use the product for ten years to come), not us, that dude!

It basically works this way. So often, we engineers get caught in a cycle of test, code, and refactoring when we don’t necessarily know that this is quite what the end user wants. Rather than test driven development, we should start from ensuring that what we are writing is exactly what the end user desires. This in effect turns the user into the test, except that this test will be the ultimate validation of our functions. Since the unit test (now the end user) is already “written” it is now the developers role to make this test pass. Once this test passes, the developers can refactor and add all the bells and whistles – thereby beautifying their code and making sure it is covered with unit tests (actual code this time).

This process allows a faster engagement between the end user and the developers. The faster feedback gives the developers a sense of what the user really needs and causes less disappointment for the developers when they have to remove all the code, testing, and refactoring into which they poured so much time and energy.

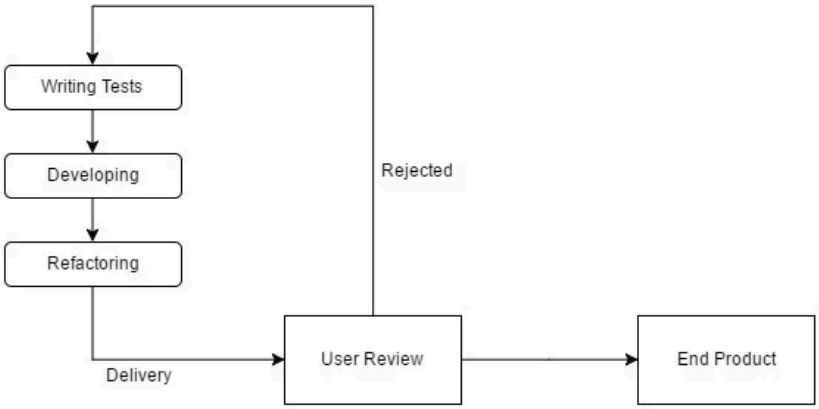
Let’s talk in numbers. If writing the unit test, coding, and refactoring costs $3000 (each stage at $1000), then that means that if the user changes his mind five times, the entire development cost will be $15,000, whereas if each time the end user changes his mind, we only do the development without the unit tests and refactoring, then the total cost is -$7000. This cuts the process by about 50% while achieving the very same end result, feature, and code.

Let’s put this in an equation:

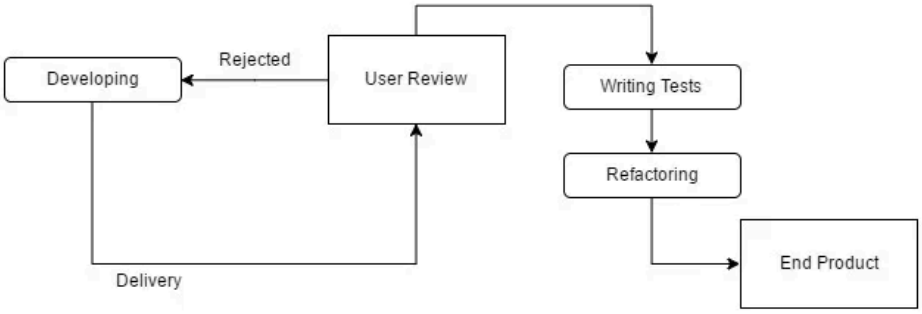
(Testing + Developing + Refactoring) x 5 => User Acceptance = $15,000

(Developing) x 5 => User Acceptance + (Testing + Refactoring) = $7,000

The following drawing also illustrates the process:



TDD Process



Optimized Process

I understand that some developers view Uncle Bob's words as scripture, but let me remind you that at one time "waterfall methodology" was the only "right" way out there. You would be surprised at how many people defend waterfall methodology and enforce it today. Test Driven Development is a great methodology to protect our software, but what if we could take that methodology (Uncle Bob) to the market and teach him a couple of new tricks – to pay less and accomplish the very same thing, maybe even more?

CHAPTER 7

Cowboys & Mad Scientists

In order to improve the software development process, it is crucial to deeply understand each and every part of it. The most important part in every software development process is of course the software engineers who turn ideas into code, and code into profitable product. This article dives deeply into what kinds of engineers are out there in the industry and how to take advantage of their diversified talents to ensure a successful software company!

In the industry there are two kinds of engineers:

Code-Driven Engineers (Mad Scientists)

Code-driven engineers were usually those at the top of their class when they graduated. They tend to be the type who focus on the ones and zeros of every popular compiler out there, and heck, they might have even built their own at some point!

These engineers are usually not very sociable like introverts, but they get delighted and very much enjoy a conversation about the newest design patterns, and what would be the asymptotic time of a recursive function verses a function that implements iteration.

These code-driven engineers are the assurance of your software safety when things fall-apart, because they are fast-thinkers in the code-verse and they will be able to pin-point a production issue and fix it before your end-users even blink.

But more importantly they are the ones that assures your software readability, maintainability and resilience against high volumes of requests/data - but they shouldn't be your visionary vanguards when exploring the market.

Product-Driven Engineers (Cowboys)

Product-driven engineers, on the other hand, are not usually the grade A type of engineers, but they are always up to date on which software is making a huge splash out there in the market. They are always reading about and quoting successful leaders in the industry, and they are definitely always looking forward to becoming one themselves.

Product-driven engineers enjoy explaining how code works to non-engineers, therefore have very high social intelligence. In an Agile environment, they are the negotiators with managers and stakeholders, always taking things even further because they are more candid and straight-forward so they end up becoming tech-leads more readily than code-driven engineer's due to their ability to effectively communicate with non-technical management types.

Companies need product-driven engineers to keep your product up-to-date with the latest patterns. They are more knowledgeable about what sells out there, they are also very aware of old routes other software companies have taken, and why these paths may have failed or succeeded and how to either avoid or emulate those steps.

Those engineers will help you find the path, but they shouldn't be the one building the final product.

The Concept

The software industry definitely does best when both types of engineers are paired together. Companies need product-driven engineers to pave the way and find the missing link for the right product that people will actually be interested in, and this requires serious experimentation. So the faster we can market the product without getting tangled up in overly ambitious code-control, the better off the software company is as a whole. The focus is kept on the product itself which is, of course, the meat on the bones of the profitability.

On the contrary, a high level of discipline is required to blow-up your entire working, profiting software and when you start rebuilding it from scratch will make it maintainable. This is where code-driven engineers come in to fill that need.

The longer I work in the software industry I realize that we can't turn one engineer-type into the other. This is something that is hard-wired into the personality or into the nature of code-driven engineers to refactor and abstract and test-drive every single aspect of their code. The same goes with product-driven engineers - they don't seem to have the patience to sit two hours through making a code that's already working, work again!

This is where pair-programming comes in to play here with an amazing result. In a pair-programming session you need someone to code and someone else to drive the development and keep the coder on track Both are very important in the process of creating good software, but there's a little bit of a role switch there depending on which stage of product development the software is in.

It All Depends on the Stage

When the software development is in the prototyping stage, you want the product-driven engineers to be the ones writing the code, and the code-driven ones to be watching, because the product-driven engineers will write the fastest, easiest thing to get something out there to the market, even though it would be hard to watch for a code-driven engineers. It is actually quite important that they actively watch to ensure the code written even if it is at this stage is very unmaintainable, that it doesn't cause a catastrophic irreparable issues such as clients losing their data, credit cards information, server failure and out-of-service nightmare.

But when the software is in the refactoring stage, now that we know people actually will use this software, you want the code-driven developers to be the ones writing the code. Whilst the product-driven engineers making sure their pairs don't go off track or waste too much time on an edge case that is 1 in a billion to happen, (in short making sure they keep the development from turning into a science project), unless they are actually working at a lab somewhere at Area 51 then this article is totally irrelevant!

In the refactoring stage product-driven engineers also make sure their pairs are not changing the product from the way it is out there "because it makes more sense this way".

Balance-First

Both are very important to achieve balance, a software development process becomes very slow when two code-driven engineers are pairing on something. They'll consume all the estimated time and there will still be no actual marketable product out there. The same goes to two product-driven engineers pairing on a feature – the code will be very hard to maintain, let go of the fact that it might cause irreparable damage to user's data, and some important edge cases will be missing because "that doesn't matter right now" or "I don't think people will use it this way" - but guess what? They will!

If the development was totally up to product-driven engineers the result would be a very poorly designed software that's almost unmaintainable, maybe not optimized to serve huge amount of data. But on the opposite side of things, if the development was solely up to code-driven engineers, they would turn everything into a science project, and nothing will ever get out of the door, because "I can make this better" syndrome.

Not Black or White

This balance is very important, and it may vary between one engineer to the other It's not always 100% code-driven, neither is it 100% product-driven, even the most product-driven engineers I have seen still had some "nerdiness" in them, same goes to some of the code-driven engineers I worked with. They still get excited to see how the market reacts to their work and become even more interested into improving things the customer didn't even ask for. When I say code-driven. I mean engineers who have more tendency to assure maintainability and beauty in their code even if the product isn't really worth that effort. These product-driven engineers are the ones who keep thinking of the end-result whilst they are still deep in the weeds of initiating their solution.

The Team

It's important for a self-aware software team to know their strength through these very categories of engineers. It allows the team to be able to manage their resources better and focus on what matters the most whether they are at the

prototyping stage, or the refactoring and maintenance stage.

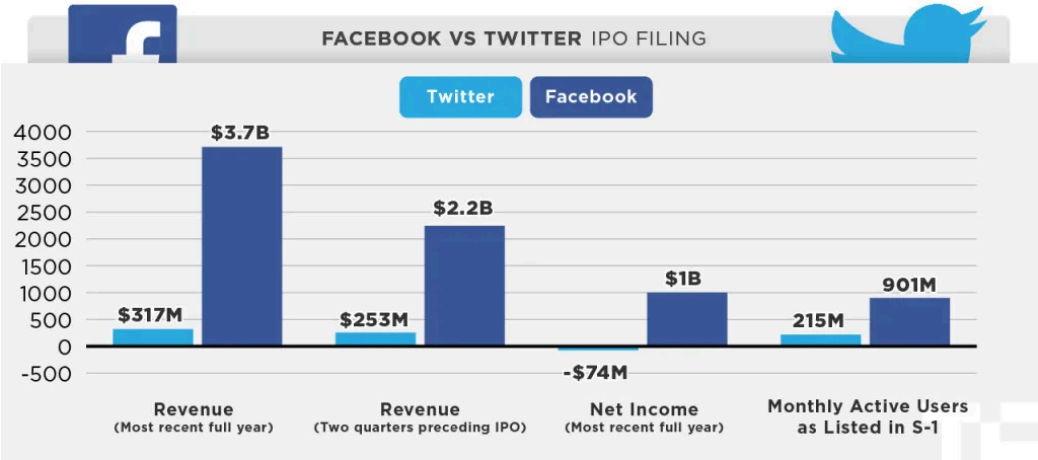
Glimpse at the Past

If you look at the history of developing software you will notice that these two types of engineers always existed in most of the popular software out there. Look at Steve Jobs (product-driven engineer) and Steve Wozniak (code-driven engineer) – Apple wouldn't have been possible without both of them, as a visionary engineer needed to be out there to know what people really would need, whilst the computer scientist would back that vision and make it possible.

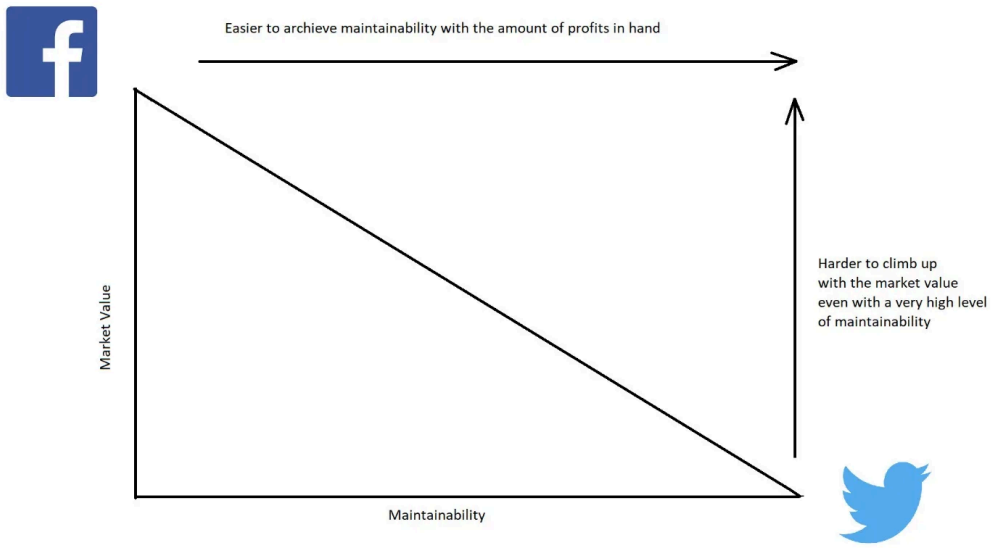
Real-World Examples

If you look at the software market today you will see these two types on a larger scale; Twitter is **the de facto for coding standard for Scala**, and the ultimate reference for Bootstrap styles, however **Twitter isn't making any money**. Entrepreneurs are still wondering how to make that product profitable.

On the contrary, Facebook is not a de facto on anything. Some of the interns at Facebook reported in many blogs that it's all just code-and-push, no revision, no testing, and **very poorly** standardized development process. But **Facebook is making a lot of money.**

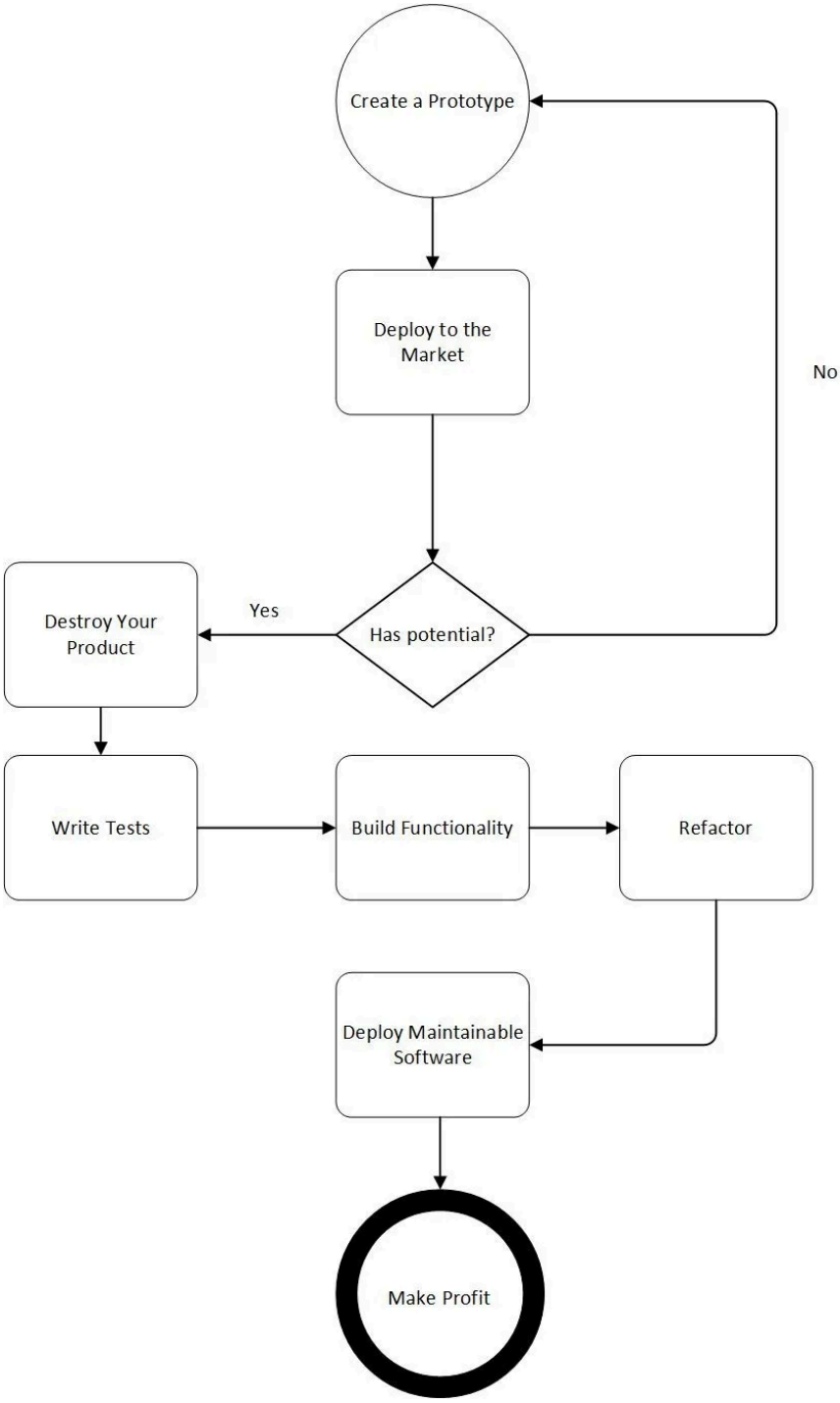


A successful software needs to be maintainable, but also profitable, but the profitability has to come first before maintainability to be able to bear the costs of refactoring and engineering. Twitter started with maintainability so they suffered large losses in market-value. Facebook started with profitability, but their code is very hard to maintain.



Facebook today has more chance to get back on the right-track (and they are) because they have the funds to rewrite their entire platform without their users even noticing that it was all rewritten from scratch. It becomes harder for companies like Twitter to get to that stage as it burns funds every second of everyday without any gains.

Prototype First then Nerd it Out!



Learning from Facebook vs Twitter example, a good conclusion would be to encourage prototyping before spending months and months in developing software that doesn't really sell, and when prototyping make sure you employ your product-driven engineers to take the lead and explore the market until they find the right product for the right market.

Knowing whether a given product has a potential or not is not done by market analysis or speculations, it's by actually hitting the ground of reality and seeing how people react to it.

No one could tell you how your product is doing like your end-user, so make sure to get that honest opinion as fast as you could.

Zuckerberg once said: "Entrepreneurial culture thrives when it is easy to try lots of new ideas, Facebook wasn't the first thing I built"

The Future ...

In the end, I think a deep understanding of a team and its abilities plays a big role in determining the success or failure of the software itself, in light of the categories mentioned above, that might be a good start in understanding how we can optimize our development process with a good strategy to meet business needs at the lowest costs without suffering from poor quality.

CHAPTER 8

Beyond Agile Software Development Methodology

Thinking outside of the box – what happens after Agile?

As more and more companies use the Agile Software Development Methodology, we are finding that while the concepts are neatly woven into proven strategies, there is much more to be gained from its use. This is a brief overview of how the Agile Software Development Methodology can and should be expanded to address not just the software development processes leading to a marketable end product but should share some of its focus on developing the developers. This gives them the chance to grow, learn, and know that they are intellectually contributing something valuable to the world of software development.

The Agile method can and does cease to benefit development teams in spite of the fact that they may follow the system perfectly. They lose motivation, they lose perspective and they lose in the market. Other companies, however, work far beyond the time or far beyond the due date on which they should deliver their products. Their team members, however, are inspired, their goals are clear, and they are reaping success in the market.

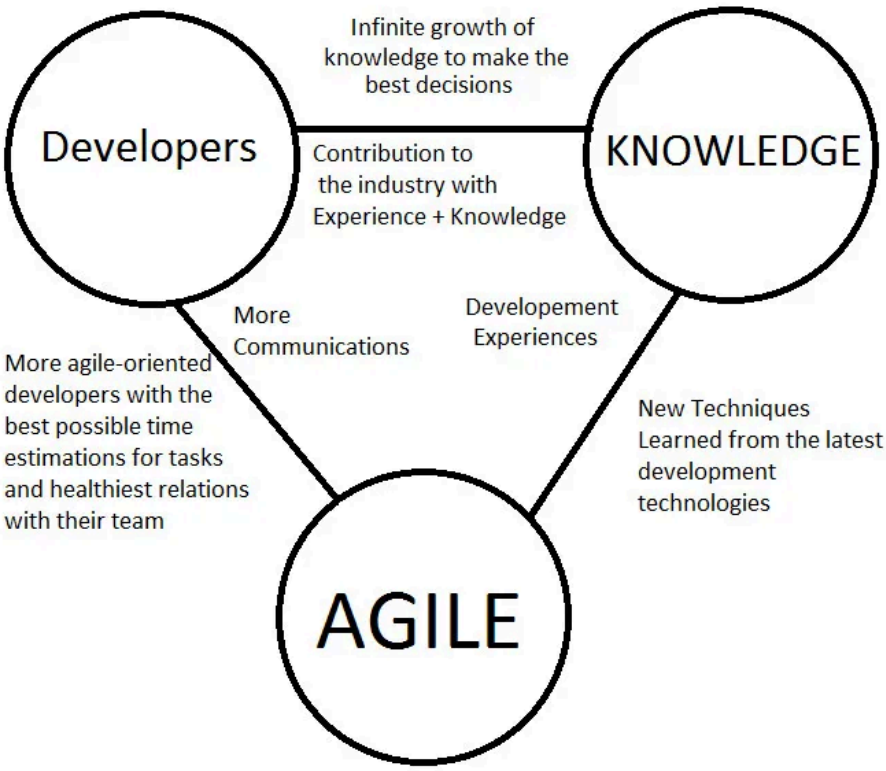
So let's look outside of the proverbial box. Agile seems to be one little block among the blocks of sprints. It is simply a method to ensure that the client and the development team are on the same page. While the bottom line is usually assumed to be financial, aside from that goal sprouts knowledge and fulfillment both professionally and personally. Programmers can easily get stuck in the Agile sprint cycle and spin their wheels unless something else is replacing financial motivation. By holding so tight to the Agile methodology, developers are quickly left behind as more up to date methods are established and new developers join the team (or on a broader scale, the entire employment market). They never become aware of what is happening beyond their scope of knowledge and expertise. They

go to work, they do their job, they get paid, and that's it. They want to do the things they already know and call it a day. Theoretically, these developers could soon be replaced by machines, much as physical laborers and assembly line workers have historically been phased out of various industries. Without staying ahead of the game – or by staying stuck in the Agile Sprint cycle – professional and personal stagnation occurs. Experienced developers are quickly replaced by each new generation of developers who have the benefit of knowing the most up to date methods. Unlike ancestral generations with many years between groups, each single year brings new graduates into the market to replace the old. It seems developers have an expiration date unless they are willing to move beyond their scope of knowledge with each passing year!

So how do we ensure that advanced and up to date knowledge is brought into the agile methodology which will, in turn, impact the financial bottom line later? The best way to ensure this is to appoint someone to gather and learn the most current knowledge and then bring it to the team. Rather than leave the individual team members to enhance their knowledge base alone, an appointed individual must take on the responsibility to learn, then teach the new knowledge to the team. By taking the time to incorporate one sprint as a learning sprint, the end result will mean a better product and a more successful company.

There is another important part to the way software development is happening. When agile methodology was first introduced, the goal was to have a team that could self-manage. The faulty reasoning behind this not only adds stress to each team member with pressure to complete each task on schedule, but more time (usually overtime) is needed in order to meet these arbitrary deadlines. This kind of self-management is really like someone being lashed on the back with the pressure to get the job done quickly and the end product suffers. It turns the developer into a machine so the missing element is the human aspect of the development process. The single most important aspect of this is to give developer the freedom to brainstorm and innovate. To be sure, the human element consists of giving people the chance and complete autonomy to innovate and this is why many programmers have moved toward open source coding. Money is not the ultimate motivator as it has been historically. Open source and self-guided innovation represents who and what the developer is at his core – not just the pay he receives from his job. No one can successfully expand if he is only worried about

tasks and deadlines. The very thing that makes each developer different from a machine is this free innovation. Rather than being told what to do, developers need to be given the broad goal and the chance to figure out for themselves what is the best and most inventive way to get the job done.



As much as we make agile sprints that focus on user requirements and the steps in development, there must be sprints for knowledge. By spreading knowledge and equalizing the chemistry of knowledge on the team, everyone is on the same level so there is consistency in the code. Unbalanced code, some advanced, other lacking, creates a mess of an end product and chaos among the team. By brainstorming the overall fulfillment of the end user requirements, the team members can find the best way to reach those end goals while not being tied to micro-managed time and creativity constraints.

When team members are set in learning (not necessarily formal education) sessions will be exposed to the latest technologies and team members will relate to how these technologies solve some of their current problems. It definitely means that tasks will get done faster, products will get delivered sooner, and the team will keep evolving. Incorporating learning sessions/learning sprints shall assure the infinite growth of knowledge which prepares the team members to make the best of their decisions.

I hope that I have shown that when developers are “developed” they are more likely to use better communication skills and make best estimations for the tasks they are expected to get done in the agile environment. Allowing investments in the developers and not just in the tasks, projects, or processes, the communication among team members opens the door for broad shared experience and the team can act as a holistically empowered organism.

CHAPTER 9

The Ping Pong Table

Tristan and Hansen are colleagues who enjoy their lunch breaks together each day as they share thoughts and have conversations about a variety of topics. Sitting in the lunch room one day, they notice a ping pong table near the coffee area.

Hansen decided to try ping pong which made Tristan interested as well. They stepped up to the table, paddles in hand, and started to volley the ball back and forth. It took some time to get the hang of it and to understand the rules of the game, but since the table was always prepared and the equipment was always handy, they wasted no time getting ready for the game. The more experienced players noticed their interest and offered advice on technique and helped Tristan and Hansen to learn the proper rules of the game. Each day they could jump right in and begin the day's match. And each day, their learning curve became smaller and smaller while they had fun honing their new skills.

Without the luxury of having a fully functioning, prepared table, Tristan and Hansen might have easily been turned off the game. If there had been holes in the net, broken table legs, or dented ping pong balls, the motivation to learn and play would have diminished. It would have made things very hard for Hansen and Tristan to play the game and it definitely would have effected their learning and skill improvement. The purpose of the game was very clear from the beginning - the minimum acceptable outcome was to simply to have fun playing, while the ultimate goal was to actually win against the opponent. And while winning or playing a perfect game from the beginning wasn't likely, (in fact it would have been surprising) ultimately it was completely acceptable the lose, yet continue to improve skill sets and overall performance.

As time went on, more players would occasionally join the game, and the purpose of the game could change from time to time. Some are more competitive, others just want to relax and blow off steam, but this doesn't affect the quality of the game. It was still fun and winning was always a worthy goal!

This story is not about ping pong :-) The end.

CHAPTER 10

Fat Apps!

What is a fat app?

An app is too fat when it takes over half of a work day (that's exactly four hours) to have one line of code changed, deployed, and made to be used by the public / end-users.

I could say that an app is too fat if it takes too long to change a simple feature, but this definition too vague, and doesn't fit the software engineering mentality. Neither does it truly serve the purpose since every single engineer on the planet has his own *version* of how long is too long, and how simple a simple feature really is.

Why apps get fat?

Apps are like some people, they get too fat when they are over-fed, except that apps do not get fed food, they feed on code and policies.

For a while I thought the solution for these bloated apps was the micro services architectures, but soon I realized it makes it even harder to locate an issue that's happening amongst these micro services without intensely logging and monitoring each and every action that a given micro service performs. Not only that, but you have to notify the entire team that something went wrong with a feature somewhere, it takes exponentially more time when the problem is located in multiple parts of the system.

So while micro services make it easy to develop a feature in a given domain, it makes it even harder to tackle a problem with too many moving parts in the system, unfortunately it's the best thing that we have out there amongst other architectural patterns.

Another reason that apps get too fat is over-engineering. Sometimes engineers try to impress each other, or “out-nerd” each other by showing their abstraction skills and their [INSERT_LANGUAGE_HERE]-Fu. When it comes to solving some problem, the result is an overly complicated app that requires more time to understand than it takes to solve the problem.

A fat app could be as small as fifty lines of code, but the governing rules around implementing features and deploying them for the end-users is what actually makes it a fat app. The “fat” here is located in the governing area of the app development process, not in the app itself, and that’s an easy problem to fix because it doesn’t really require technical engineering. It becomes more of social engineering solution (in a good way) to change how things work around the app delivery processes in which case the actual complexity here lies in the flexibility of the governing team to change their methods of deployment and certification of a given app.

Solutions

Just like most of the overweight issues, there are multiple things one could do to avoid producing fat apps. The first of these solutions is exercise - not the exercising one does at the gym, but exercising simplicity. Every so often we engineers work so hard on learning the latest and greatest in a given technology, or a development methodology, but very seldom do we exercise simplicity. Simplicity is an art form of solving a problem without overthinking, or over engineering a solution.

Simplicity, however, is not another definition for readability either. A piece of code could be very readable, but it may still be very complex and over-engineered. Determining if something is simple, or not is so easy, a team needs only to let the most junior developer on the team try to understand and explain (maybe even make a simple change) the given implemented feature, and if it takes them over an hour to understand something, then that’s your fat app sign right there. In this pattern, junior developers on teams are just as important as the senior ones because they are one of the scales that developers could use to determine if their app is overweight or not.

Another solution is to be realistic about implementing apps. The act of not worrying about what *might* be needed in the future (and only focusing on the now) creates a dilemma. The "now" are the requirements in hand which should lead to less abstraction and more simplification toward what a piece of code is trying to do.

Too many team members = fat apps, or too many cooks spoil the proverbial broth. Since each one of us has his own *right way* to get something done, we may end up in a middle ground amongst the team members in a situation which they might agree on a solution, but that solution isn't necessarily a simple one. Again, in this case the junior developer on the team should be the judge.

If everyone on the team happens to be a senior guru, then extracting the solution into a flowchart and showing it to a non-technical person might be a great way to determine the complexity of the app. Normally, non-technical folks are pretty honest about how *nerdy* something is when it comes to looking at flowcharts. You may also pay attention to their eyes as they look at the ceiling or checking their phones, in addition to their yawning as you to explain something to them.

Flowcharts (the dead lost art of visualizing a process) are pretty self-explanatory when it comes to indicating the complexity of a given app.

A look to the Future

I personally still think there's a big chance out there to come up with a better architecture pattern to assure simplicity without having to bear the costs of software development, but I also still believe that apps are a reflection of our thought processes. If we view things in a complicated way, this will be reflected in the apps that we make. Our apps, after all, are like our children - they only know what we teach them!

The reason I believe this is because the most complex "app" we use on daily basis is this universe around us - our bodies, our surroundings, and this infinitely expanding universe around us. With all of its complexity, the universal system is still built out of very simple molecules that effectively communicate with one another into bigger systems. This should be one very strong source of inspiration when it comes to offering intelligent yet simple software.

CHAPTER 11

The Fish Mentality!

Each and every one of us at some point in our careers feel the need to change their current job, either by moving on to another opportunity, or by simply retiring. This article is about the folks who want to move on to another opportunity. In my experience going through so many contracts and permanent positions so far in my career, I think I have a good handle when giving advice about changing jobs.

We - as human beings - are different than fish because we learn from our own, or others' mistakes. Fish on the other hand, since the beginning of time, still fall for the very same mistake much to their demise, by eating the bait.

Again, as human beings, we share our experiences with products, people and companies to fulfill a natural urge to not let any other human being fall for the very same painful experience we had. However, there are still some of us today that do indeed have that fish mentality. They fall for the very same mistake someone else made, because they don't do their research about where they are going before stepping out of their current position, they just follow the high rates or the benefits, in other words the bait!

So, as someone who has been in this predicament before, here are a few of my tips to help you make a good decision before jumping quickly to another job.

Glassdoor

this website is basically a place to anonymously share a certain experience one has had with a certain company. This website comes in very handy and people are voluntarily (and most of the time very fairly and honestly) share their experiences at a certain company. So keep that in your favorites list in your browser. You may not need it right now, but you may in the future and I'm sure a friend or family member will.

LinkedIn

this network has so much gold if you dig enough, for instance, LinkedIn will tell you the percentage of employees leaving or joining a certain company, so when you see a given company's employees are getting fewer and fewer every year, that's a bad sign, but not a deciding factor, so do not make it your only sign.

LinkedIn also allows you to search for people who worked for a given company before, you can simply search and have a personal message to ask someone about whether you should join or not. Usually people are very helpful when it comes to asking them about previous experiences, if you ask the right questions, for example: would you go back again to work for this company? Or something like - if I were to choose between this company and another, what would you recommend? Questions such as these can be very helpful.

Google

of course, if you put a given company's name between double quotes "SOME COMPANY" Google will specifically find every single mention of this company all over the web. People tend to leave very informative comments on forums and public mailing lists that you may not find in Glassdoor or LinkedIn, so this is an important resource too.

Recruiters

Recruiters are an infinite source of information when it comes to where they recruit, they know the ins and outs of every client, candidate, consultant, you name it!

recruiters will not lie to you if you ask the right questions - a question like, "have you recruited for this client before and is the consultant still there?" Inquiries voiced this way could unveil some very interesting insights about whether you should make that move or note.

Recruiters make it their job to gather facts and are full of information about certain clients and consultants. If you ask the right questions you will get very interesting answers.

Your guts!

follow your instinct - go into the company and ask to spend a day there, walk around and talk to people, see how you feel, most of the time your gut won't lie to you. It's very easy to sense tension in the air or some pretentious attitude if you are observant. Even the most professional "actors" can't stay in the "role" for very long, so experience the place and find what you feel.

Friends and Family

explain your situation and whys and the what's of your desire to leave a certain place and join another Oftentimes, a friend or a family member might know someone who has been in that place before, and maybe they are not on LinkedIn (yes, that's possible) or Glassdoor, but they may give you very interesting insights about the place and whether you should move or not.

Friends and family also love you for who you are, so they will purely give you advice because they want what is best for you and your happiness, so take their opinions very seriously.

"Life is short, work somewhere awesome!" Some dude on the internet

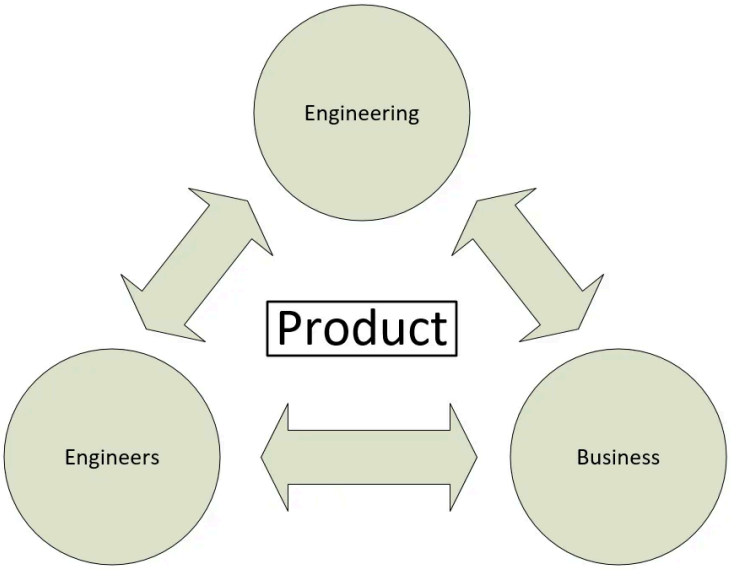
CHAPTER 12

Software Doctors

How to determine if your software development business is not healthy? And what can you do about it?

What do we mean by an unhealthy software business?

This means the business is getting dangerously close to a point where the entire business is going to come to a halt. Ways that this can happen include going bankrupt, having software that is too complex so that no one wants to learn it and improve it, or the business is unwilling to allow its engineers to improve it, or the engineers are quitting because they are not growing (and I don't mean from the perks!).



The Business

An unhealthy business could be caused by multiple things.

For this article, I'm going to focus on the profitability of the business and how that relates directly (in most cases) to the successfulness of its software as the most prominent sign of whether a given software could prosper in the market or not.

Sometimes, a business might get too focused on the software development process and forget about the profitability of the business itself.

For instance, Twitter is considered the de-facto of software when it comes to the standards of Scala, but it's profitability isn't as large as it's effectiveness in the technology itself.

This is a big indication that the business, its software, and its engineers haven't reached quite the right balance, which results in an unhealthy business overall.

A good marketing strategy and business study and financial management should take care of the business profitability part without going into too much detail about how to handle these issues as they differ from business to business.

The Engineering Process

The other symptom is the software process itself.

The diagnosis of an unhealthy software development process is simply a process that requires too much manual work of what could be automated, or it requires a lot of time (think unnecessary meetings and too many levels of decision making) before reaching a decision to explore a certain feature.

This results in the developers losing interest in participating in the process as it becomes a mind-numbing process which defies the entire purpose of writing software (automating something that used to be done manually in any given period).

Another symptom is the complexity of the process itself even if it was automated, a bad architecture or the process of pushing new features or fixing defects is time-consuming activity, or the code itself is complex (as in hard to understand and

maintain). In such a case, revisiting coding standards, introducing better architecture and changing the process of pushing features and code reviews is a must.

The Engineers

The third most important aspect to consider when determining the health of any given software business are the engineers involved in writing the software.

A lot of businesses consider taking care of engineers is only the base salary and the perks.

But engineers become more fulfilled when the business invests in their growth, from an educational perspective.

In software, the best way to invest in an engineer's growth is to have him get involved in practical experiments where he gets to try to learn about new technologies and become a storyteller (as I call them) about how to use a certain technology works for a certain business purpose.

Judging from my personal experiences in the last decade or so, I have seen a lot of businesses that invest so much in the perks and salaries and benefits for their employees, but they don't understand that software engineers have a whole different set of growth mindsets, which go beyond their basic needs.

Think of it this way, anyone can provide a salary, there will always be this other "shiny" company that provides better perks and better benefits, but only few will invest and encourage the experimentation and growth as a fundamental part of the business itself.

When the business invests in its engineers it will automatically leverage the software process. As the more educated and experienced engineers become, they will become more intelligent in building a more efficient software development processes.

More so, engineers will eventually start contributing to the business by giving a different perspective on how to tackle a certain business need such as attracting more customers to the business through software.

When the engineers are happy, that'll form an ownership mentality that will have them do their best to make sure their software (their baby) continues to grow and flourish which will eventually encourage them to provide some input to the business which will hopefully result in a more profitable business.

At the end of the day, any software development activity, let it be business, process or engineers-related that doesn't directly benefit the business, the process and the engineers involved would be considered a waste of time for everyone.

CHAPTER 13

Mr. Sam!

Every once in a while, computer scientists come up with a new architectural model that inspires developers to build better software which has the potential to reflect positively on each respective business.

Throughout my career, however, I have observed an architectural model that seems to form itself within organizations when there is a complete lack of or small degree of discipline surrounding standard software architectures. I like to call this the "Savior Architectual Model" - or Sam for short.

The savior model is simply the kind of architecture that is so convoluted and complex that only its maker can understand it, hereby known as the savior!

For others to be able to modify, or even attempt to improve upon this software they must consult the savior and get his or her blessing before contributing to the code in order to get any tasks done.

So when things do go wrong, only this proverbial savior can save the day and hence, get the usual praise for "making it work".

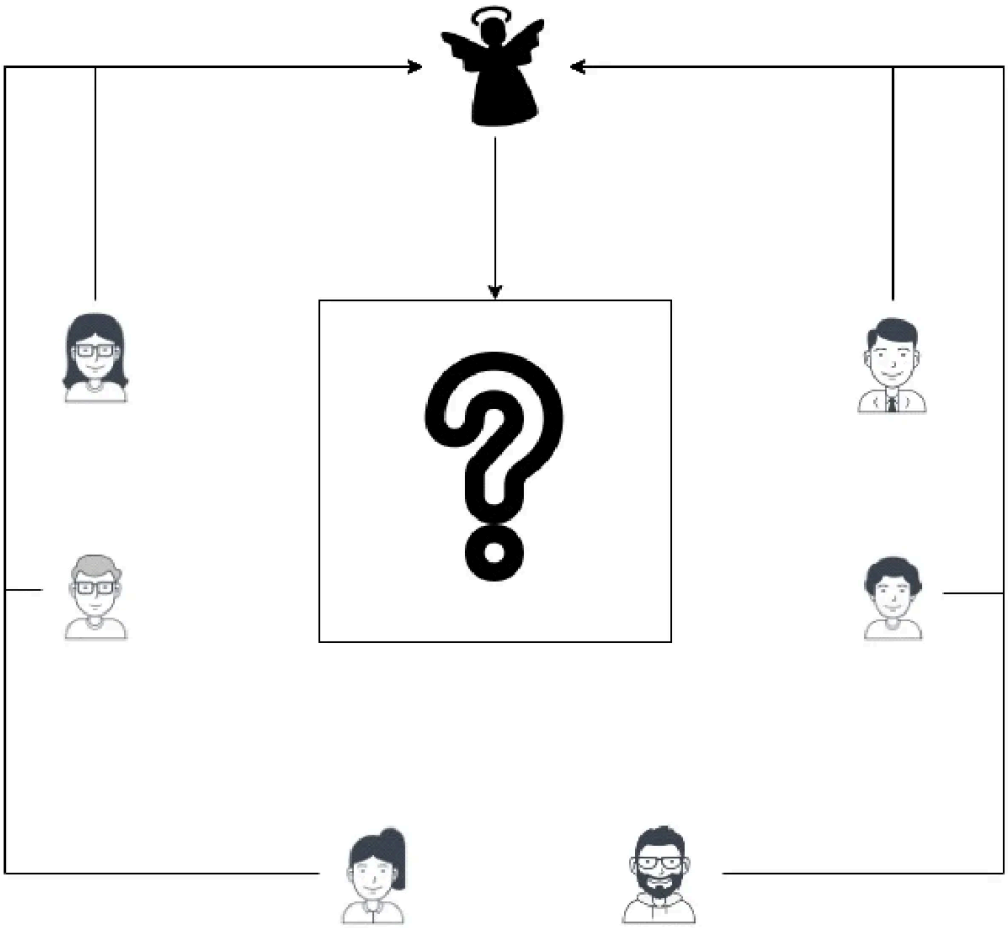
Saviors come in all different shapes and colors.

Some are just grumpy, some may hate their jobs, some are neurotic, some may be clinically depressed and some others just don't like people.

But it seems it's always your fault if you don't know how to communicate with them!

Forget about your computer science degree. Who needs that? You must have a psychology degree to be able to get some sense of "documentation" (heh, what's that?!) or be able to write a single line of code without being bombarded by criticisms from the savior's followers.

You must make sure your code change has a comment that says: “I have been blessed by the savior!” in order for their followers to leave you alone and allow you to “innovate”.



The savior model happens when organizations don’t stress the importance of involving multiple parties in the design and decision-making process.

It also happens when organizations loosen up on documenting how any critical software is being built, and what architectural standards are being followed to develop the software.

Especially the second part, as software teams change all the time, you might end up with one person on a team who happened to be there when all the “magic” happened. And that’s how saviors are born!

Outdated architectures, like in the monolithic applications, are a good example to inspire creating all kinds of saviors.

When the code base is tremendously huge, and there are few or non-existing tests in addition to the lack of documentation, saviors start to creep into the organizational structure and slowly they become a bottleneck in the production process.

As teams grow around a monolithic application, so grows the number of requests to submit new features or fix defects which requires more questions and more understanding of how the system was built to follow some sort of "pattern".

And here comes the issue of having to return to the savior to answer all these questions and explain how the mysterious software works.

This adds to the other bigger problem in which your software is running into issues and only the savior knows how to solve them. Now imagine if that happens on your savior's vacation day? Maybe at midnight? While they are being hit by a bus?

Your software comes to a halt and you lose more money than what you would've spent enforcing documentation, education and ownership across the entire development team.

Software development teams need to press very hard on few things to avoid falling into the savior model:

- Follow global architecture standards so newcomers to your team are easily familiarized with the system and are able to go search the web and read books about what standard the system design follows so they can contribute with knowledge and not break your software.
- Stress the immense importance of documentation and rotate tech leveling sessions to continually discuss and educate your team about how the software works and engage them in the design process to create a sense of ownership and pride in the software they're building.
- Unit test, acceptance test, integration test and system test your software. If your software is tested properly with high code coverage, it's very likely that engineers will figure out somehow what the code is meant to do without having to ask too many questions.

- Adapt a new architecture pattern like the Microservices architecture to reflect your organization communication structure and distribute ownership across teams and team members. Smaller code base and smaller teams inspire ownership.
- Rotate development tasks on the development team members. Don't have a "go-to" guy for doing a specific task, that's how you create mini-saviors that eventually turn into saviors that slow down and silo and tribalize your software.

At the end of the day, allowing the savior model to creep into development teams gives too much power to one member - the "savior" and leaves the rest of the team in a prison of thought that doesn't give them much room to innovate and have a counter equal input in the design process of the software and leaves them intellectually paralyzed which should inevitably negatively affect the overall performance of the organization and the software being written.

And if you think giving one person too much power to make decisions doesn't necessarily corrupt them check out the **Stanford prison experiment.**

CHAPTER 14

How to Effectively Connect to Each Other?

Having to work with a largely diverse group of people and travelling around the world, I have learned a lot about us as humans, especially being an extroverted person myself. I was always curious by what one could do to make a connection to another human being, especially if they have a different background, culture, country or creed.

Connecting to each other is a key to our growth holistically. When we give up trying to connect to one another, societal diseases start to emerge, such as misunderstanding, gossip and xenophobia which eventually turn into real physical problems and laws of discrimination in our world.

I strongly believe that somewhere in the core of every problem we have in our world, regardless of how it manifests itself, there's some deep miscommunication that has been caused by the growing problem of not knowing how to communicate and connect to one another.

Some organizations try to mechanize our communication, and this takes the heart and soul and authenticity out of it. We have to remember that the very thing that makes us interested in one another is our humanity and when we try to computerize our communications without considering all the three aspects of ourselves, we end up with what is known today on the internet as the trolling phenomena where people will say things without considering the outcome because even though we call it a "connection online," it's completely stripped of all aspects of our humanity.

So, what are these aspects of humanity?

Physical Aspect

I started with this aspect because it's the easiest aspect to observe and in some cases it's a decision maker in our day to day lives, sometimes justly and some other times very unjustly.

The physical aspect of our humanity is the idea of what we can see, what we can move around, and what we can go to from one point to another.

Some of us are very physical human beings, like mechanics, construction workers and every single person you've ever known who says they'd rather do roofing in July than sit at a computer all day.

When we try to connect to a person who is predominantly physical, it's important to understand that they want to see a picture through your discussion about physical things, moving from point A to point B, doing physical things that result in more physical things. An example of this is the way the physical act of construction leads to the physical entity of a building.

Even if the topic of discussion is about an aspect that is not physical, such as talking about taxes at the end of the year, it's important that the physical aspect is represented when you're trying to get the point across to a physical person. For instance, talk about moving funds to a certain bank account, rather than talk about how these funds were calculated and accounted for.

Intellectual Aspect

This aspect revolves more around thought, logic and rationalization of everything. It's the idea of trying to make calculations and decisions before taking a physical action.

Intellectual people might panic if talked to about taking an action without thinking it through, and for the most part the spiritual aspect might be a side detail that is a nice-to-have, but not crucial in any achievement. This is simply because a lot of intellectually predominant people don't necessarily believe in the spiritual aspect of ourselves except what comes about through logic and action - those things that are observable.

When engaging in a discussion with an intellectual person, it's very important to be very precise and simply on point. Do your best to quantify things based on their observable impact in the real world, otherwise their most predominant aspect will reject your discussion or lose any interest or neglect to give you any attention to what you have to say.

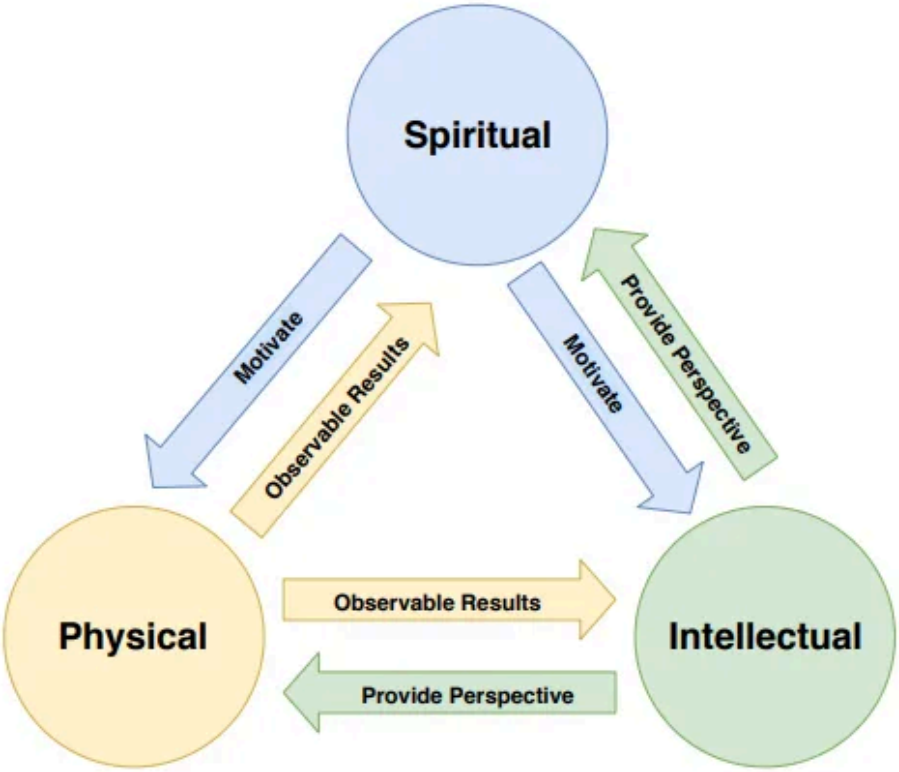
When speaking to an intellectual person about a spiritual topic, it's paramount to stay grounded to observable, verifiable aspects of life that they can rationalize. For instance, if you want to talk about happiness, try to link that to success which is rooted in hard work that can be measured and calculated by setting up goals, following regular routines ... etc.

Spiritual Aspect

The spiritual aspect is the core of our humanity, it's the unobservable force that moves us into one direction or another, the feelings we experience on daily basis from love to fear, happiness to sadness or anger.

Spiritually dominated people are like the oil to the engine in your car, your car could have the biggest engine and the smartest tech, but if there is no oil, there's no energy to move your car.

When talking to a spiritual person about a physical subject you must engage ideas such as hope, and happiness into your discussion. They are not so much interested in the technicalities of things, neither are they as much invested in physicality of what happens. They just want to feel happy, fulfilled and excited to engage into any aspect of their lives.



Each one of us has all three aspects within ourselves. We engage on daily basis in physical activities, things as simple as driving our cars or taking the trash out.

We also engage in intellectual activities, discussions, calculating our budgets and deciding which route we need to take to go to work to avoid traffic today.

We also engage in spiritual activities such as loving to go back home and see our kids, getting angry that our favorite basketball team isn't performing as well, and we get scared when the lights go out and we hear a noise in our backyards.

The tough part is to be able to identify which aspect is predominantly the strongest aspect of the person you're trying to communicate with and connect to. In some people it's tougher than for others, simply because of the economics of our lives and because every aspect doesn't have enough opportunity to engage itself in life as the others, so some people try to override their most predominant aspects and work jobs they really don't like so they can live better ... but they are not truly happy.

I learned recently that over 50% of the workforce in the United States hate their jobs. And while a part of that has to do with how far they wanted to take their passion and feed their strongest aspects, it also has something to do with the economics and diversification of our workforce from a humanity structural perspective. In any given team or within any given corporation, management should recognize the humanity structure of each employee and place them accordingly.

Being able to understand and communicate with one another is the first step into an optimum distribution of our strength across a smaller team, a startup, a bigger corporate or an entire country.

The perfect conversation with someone is a conversation that has just as much ratio of every aspect as the person you're talking to, and while that is very difficult to achieve it is not impossible when we get to understand each other more.

That's why it's important for us to listen to each other more and understand our interests and most dominant aspects before we start talking to each other. It's important to get to know each other better although personalities also vary when it comes to how fast we can detect what makes the other person more comfortable to talk about from these three aspects.

It's also important to stay authentic to ourselves and express ourselves as clearly and as honestly as possible to help the other person understand who we are, and what aspect we are more drawn into that connects to us.

We engage in conversations every day, with our family, co-workers, and strangers at the coffee shop and everywhere else, and if we improve how we communicate with each other, we improve each other's lives reciprocatively as a whole, which should eventually reflect positively on us, our families, friends, society, and the rest of the world.

CHAPTER 15

Time-Driven Engineers

Through my career, I've come across different kinds of engineers that I've learned a lot from. But indirectly I have also learned from them the different patterns that directly contributes to their professional growth and their ability to stay relevant in this industry.

There are two very observable types of learning patterns amongst software engineers, I like to call them in more technical terms as time-driven engineers and event-driven engineers.

Let's talk first about the Event-Driven engineers (EDE).

EDEs get only motivated to learn and grow their knowledge when an *event* occurs, such as having to do an interview, moving into a new team and having to learn a bunch of new things to catch up, or simply having to do a tech leveling session to educate others about some new technology.

The problem with being event-driven, is that events don't occur that often for most people to continuously learn and grow, which results in the case of stale knowledge, especially in the software industry, what you thought you've known is now old-school and v 2.0 is out and you got to keep up.

An EDE could spend over ten years or even more working at the same company with the same domain knowledge and the same systems without having to learn any new skills, technologies or patterns. Merely making sure the system continues to run as expected and that's it.

The other kind are the Time-Driven Engineers (TDE).

TDEs have a continuous scheduled pattern of learning, everyday represents an opportunity to learn more, grow more and to take advantage of the time they have available through their day to get to make the best of it.

TDEs can learn in so many ways, such as reading blogs, code, books or watch tutorials or simply have daily meetings and joining groups where they discuss different technologies and patterns.

But teaching others can also be an opportunity to learn, we retain over 70% of the information we teach others and it helps us get different perspectives about what we've learned when we communicate that knowledge to others.

Being a TDE could cause missing opportunities to learn from certain events, being a EDE as we said, could cause a stale outdated knowledge and getting out of touch from what's going on in the industry.

The best of the two is to be both, continuously schedule learning new skills, patterns and technologies on daily basis, but being able to adjust and be agile with your learning schedule when an interesting event arises to learn something from.

Learning because of certain events occurring is very beneficial if mixed with continuous learning and growth, sometimes some events occur that will consume our already scheduled time for learning so we can learn from them and be able to handle them.

And that's fine, we should be agile and flexible enough to adjust our learning schedules and patterns as we go – otherwise learning would be very academic and dull if it can't be connected to real life and be adjusted based on the events that we go through.

Trying to adjust my continues agile learning schedule, I found it to be very beneficial to change what you learn and how you learn it every day, for instance as a software engineer, it's very refreshing to take some days to learn simply by reading someone else's code on github, while other days I'd read books, articles or blogs and some other days I'd just watch tutorials, videos or listen to podcasts of interesting topics around technology.

And like I've said earlier, teaching others and getting feedback from them could also contribute to our own learning, experience and growth in soft skills like communication skills, positive attitude and self-confidence amongst many others.

Switching between the ways we learn could have a big positive impact on the continuity of our growth. Having several ways to learn plays much better with how our brains are built and eliminates the feeling of dullness or boredom that may occur from following the same pattern of learning on daily basis.

Here's some of the different ways we could learn as engineers on daily basis:

1.Reading (books, blogs, forums)

2.Watching (tutorials, conferences, lectures)

3.Discussion (With colleagues, friends, family, experts on social media, emailing authors)

4.Teaching (Online, mentoring, lecturing and writing blogs, answering questions on StackOverflow, sharing articles, thoughts and experiences)

5.Coding (Open source projects contributions, reading others code, engaging on Github in documenting and discussing pull requests)

And finally, we should treat learning the same way we treat our basic needs such as eating, drinking, and breathing, it has to be a fundamental part of our day to day lives, feeding the intellectual aspect of ourselves is just as important as feeding the physical and the spiritual aspects to achieve true happiness through success, and harmony within ourselves, our families, and friends and with the rest of the world.

CHAPTER 16

Foreign Code

When we start working on real software projects, it's very likely that the team you'll be joining will have an existing source code rather than starting a project from scratch.

Every source code comes with its own set of technologies, patterns, practices, and policies, and that's when we run into this problem of not knowing where to start or how to understand what this code is doing especially if it's a massive source code.

Often times documentation will not be available, and if it is, it might be outdated or high-level design that may or may not be fully followed or implemented across the entire project.

The only actual and truly reliable documentation you have is the code itself, and possibly the people who wrote it (if they are still around) – everything else is at best a 50:50 chance.

So, if you found yourself in a situation like this, what can you do to understand the project you're working on? How can you minimize your learning curve and maximize your contribution to the project?

There are a few things I have learned in the past 15 years that I'd think might benefit you in situations like these:

Domain First (The Big Box)

The first thing you should do, is invest most of your time trying to understand what business is trying to achieve through the software. Keeping this overview in mind will help immensely.

Understanding the domain will help you pinpoint where the code is headed as an end goal, which will tremendously help you determine which of the existing code can stay and which of it can be completely removed from a high-level perspective.

This should play a big role in decreasing the complexity of the code and will make it much easier to understand.

Understanding the domain will help you regionalize the code, that is, being able to see bigger regions of the code as big blocks that you can move around and see the relationship between each and every one of them.

Architecture (Lego Pieces)

Along with the domain understanding, you should focus most of your efforts on seeing what flow the code is following in order to accomplish its goals.

Some source codes are harder to understand than others from an architectural perspective, especially if the architecture isn't very familiar or common. Your best bet would be finding the architect that originally designed the software to learn from him or her in terms of what the original intent may have been.

Understanding the architecture will help you greatly in knowing where the code is following its original design and where it went off the rails and violated its destination to get something done.

Being able to see where the code is violating its high-level design will help you see the cohesion between the components of code that actually follow the rules, and it will give you even a better idea of how to bring the off-design components back into the original design, which should eventually contribute to the quality of the source code and the project as a whole.

Allocating the time for refactoring and rebuilding existing components that "just work" is a whole different topic that I have discussed in a different article.

Patterns (Smaller Lego Pieces)

There are so many different patterns to write software, creational, structural, behavioral and others.

Understanding what patterns the code is or was trying to follow will help you see on a class-to-class level how features are intended to be built in the code.

It will help you write a cohesive code that follows the agreed-upon patterns which will eventually result in a lot less code review comments and a lot faster ability to deliver features and resolve defects.

It will also help you see again the parts of code that violated these patterns, so you'd understand which is meant for consistency and which is crying for refactoring.

Coding Guidelines (The Nitty-Gritty)

In most teams, you might come across a coding guideline, a set of rules the team agreed on to follow to create a cohesive class-level code that is easier to read and understand.

If the team you're joining doesn't have a standard coding guideline you should reference the universal coding guideline for the programming language you're using and make sure everyone on your team is on the same page.

A cohesive code that is readable plays a big role in your understanding of where things are going from a low-level perspective.

Someone might ask, where do we get all that information about the domain, the architecture, patterns and guidelines?

There are few methods you could follow and I recommend diversification of these methods in order for you to get all the information you need to learn about a certain project:

1.Find all possible documentation, even if it is outdated, and build a knowledge-base around common terms, abbreviations and concepts your team is using to refer to certain features in the system.

2.Once you build your knowledge-base, you can now communicate with the program managers and engage them into domain discussion about what the software is trying to do from a business perspective.

3.Communicate with the architects and team leads about what high level architecture the software is following. Then go read or watch as many tutorials as you can about that architecture and its best practices, and then engage in

discussion about what you think is violating the standard architecture to get a better understanding of how the system is designed.

4.Be sure to communicate with the developers and engineers on the team to understand what patterns are followed and what coding guidelines are enforced to maintain the code, then again, go back and study about these patterns and run comparisons to find where it best fits to follow the pattern and enforce the guidelines.

5.Start doing some clean-up work and fix potential defects to get a deeper understanding on the code-level to grasp deeply how the team communicates their thoughts through code reviews and fully understand what they recommend for a better clean-up work procedure.

Refactoring is always a great opportunity to engage the team in a discussion of what the best practices are and what guidelines should be revisited and what others should be enforced.

6.Share your findings and your documentation with the team, gather them in meetings and continue engaging them in constructive discussions about the business, the architecture, the patterns and the code itself. Doing this regularly doesn't only contribute to your knowledge, but it also levels the knowledge across the team, because there's always this one thing that this team member didn't know about, even though they've been around for a while.

Following these steps doesn't just help you understand the new code you're working with; it is also the best way to introduce yourself to your new team, learn from and with them and work for them to achieve a higher level of success.

There will be situations where you'll run into blockers, people that are harder to communicate with, job security issues that might impede you from understanding the system, and in that case the best advice I could give you is to try to find other doors to get to where you want to go.

And if there are no doors, then go ahead and create one. Don't give up too soon and know that reviving a dying system will bring you more experience than working with a perfectly designed one.

CHAPTER 17

Would You Hire Your Manager?

Staying around in an organization long enough to get promoted into a lead position is one of the biggest mistakes some people in corporations do.

Poor leads who only got into a position by staying around long enough could be the cause of a catastrophic cultural disaster within an organization. That doesn't just hurt the company's reputation and causes a high attrition rates, it also has a ripple effect in an interconnected world that could cause an entire organization to crumble.

That's simply because leadership requires more than just experience and knowledge of the business domain and the technology.

Leadership requires special skill-sets in addition to the two mentioned above.

On social media, you will see a lot of quotes about how a leader inspires people and motivates them, how a leader needs to put the interests of his team before his own and many more like that. These are great, but seldom will you find an actionable road map into how to become a good lead in the tech industry.

Having the opportunity to work as a tech lead for a while throughout my career, let me share what has worked out for me, and show how each has person his or her own ways of finding the best way to become leaders.

Looking closely at leaders I've worked with, read or heard about, I've come up with few fundamentals that are the must-haves for a lead to become a good lead. Here's my list:

Leniency

If you want to build a team of strongly loyal engineers, you have to be lenient. Leniency comes in so many different shapes as you can see:

- **Be easy to talk to**, the easier you are to talk to, the less you'll be surprised by how your team members feel about working with you or on the project.

Being easy to talk to inspires transparency and honesty among your teammates which will eventually result in growth. It will also help you easily identify what you're doing wrong and be able to rapidly correct it before you lose your team.

Being easy to talk to also comes by being the person who is consistently listening to your team more than talking, trying to read between the lines about their problems, listening closely and very carefully to what they are trying to tell you. Not every person is direct and comfortable talking to people in general, especially in this industry, so being the one who is, goes a long way.

- **Be understanding** of people's circumstances, put yourself in their shoes, help them through their tough times and encourage them along the way. Believe in people because when you do, people will come through for you, for your team, and for your corporation.

Forgive their mistakes, and help them overcome them, and learn from them. Don't give up too soon. Be sure to try once, twice, thrice ... whatever is enough to build an informed decision of whether your team member is actually trying to learn from their mistakes and focused or not.

- **Connect with your team**, learn about their interests and share your thoughts and your own interests. Find common things between yourself and your team. We learn about new technologies when we move to new projects, but we also need to learn about every team member's interests and explore them with them. This is a bridge-building mechanism. When you do that, people feel more comfortable working with you and sharing their concerns with you.

Protectiveness

A good lead protects his team, has their backs and provides a safe environment for them to voice their opinions and supports them. This protectiveness comes in several ways:

- **Internal Conflict**, when two team members have a disagreement, you need to step in and protect each of them from losing communications and the willingness to work with each other by trying to bridge the gap and clearing any

misunderstandings that may have led to a conflict. The best way to clear out internal conflict is to create a research session by gathering everyone in order to engage all team members in the research so everyone could grow and learn together.

- **External Conflict,** you have to have your team member's backs when engaging in a conflict with an external team. Work with your team members to investigate the issue. If your team member is mistaken, own the responsibility entirely as a team and use terms like: "We have had an oversight over this particular issue ...", language is key to maintaining an effort of growth with the least amount of damage as possible.

When you face the fire for your team, you inspire them to research more, grow more and learn more before engaging in discussions with others. You will then create this internal need for them to make you proud and honored to step into the fire to protect them.

Inclusion

Inclusion is a key to maintaining a good relationship with your team. When you include your team in your decisions, they take part of the responsibility and ownership of the outcomes of that decision.

Be fair and give everyone an equal chance to vote for what they believe is the best way to go about tackling an issue.

Inclusion sends a subtle message to every team member that their opinion matters and that they are trusted to influence the direction toward where the team is going.

Inclusion also inspires growth and accountability. It encourages open discussions, more research to make a better decision, and more communications. It offers more opportunity for your team members to communicate and the closer they get to each other, the stronger their relations become which shall eventually result in a better culture and a stronger team.

Inclusion is favoritism worst enemy, the less inclusion you have on your team, the more favoritism will start to creep into your team, which shall result in isolation, silos, the optics of levels and other cultural diseases that eventually leads to a failing team and a failing organization.

Software is only a reflection of our communications, if we are closely connected to each other, the software we build is going to be just that. The further away we silo ourselves, however, the worst the product we build shall become.

I've seen some people who think being tough and intimidating is a good thing to have as a leader. Historically speaking, those fear-inducing leaders either ended up with minions around them, robots who have nothing to say or opinions to make, which eventually brought high stress on these brutal leaders than they could handle and led them out of their organization.

There are also the intimidating leaders who ended up being alone, with nobody to support the software. New graduates will join their organizations but soon enough they leave, because this is not the 1990's anymore. People talk, communicate, and read articles like this one that'll hopefully inspire them to do what is best for themselves and take their careers in the right direction.

CHAPTER 18

The Lifecycle of Technology

How is technology born? What makes some technologies so popular while other don't make it past the drawing board? Why do some technologies get all the way to the top, then die far too soon, when others live on for decades?

In this article, I'm going to discuss the life-cycle of technology, be it a cellphone, an electric car, or just a simple programming language or development pattern. All of these (and more) fall under the definition of technology.

There are several key players that contribute to the life and death of a certain technology. Technology is created, made popular, consumed and destroyed by people – from the smartest technologists to the average consumers.

The Visionaries

very technology out there starts with an idea that's driven by a passion to right a wrong, or change the world as we know it into something better. The best technologies out there are the ones born out of the mind of a visionary who is trying to make the world better by starting with his or her own life, wanting it to be something more than ordinary.

Visionaries are not necessarily engineers or technologists, but they are futurists who try to imagine and envision a better, faster, more efficient future, but then they have the courage to take that to the next level and turn an idea into a reality.

Visionaries don't necessarily come up with an idea that was never thought of before, they might take an existing idea and do it differently, or do it even better!

There are many visionaries in our modern age and we could certainly learn from their stories. People such as Steve Jobs, Jeff Bezos, Elon Musk or Bill Gates definitely come to mind.

It's true that each one of them might have had a different story of how they found their original idea and took it to the world and changed it forever, but what we should seek here are commonalities among these people, starting from where they stopped and become visionaries ourselves.

One of the most fascinating commonalities among all the visionaries in the past and in our modern age is their courage to take risks and try different things, especially when trying to make something our society doesn't yet understand. The benefits of a new idea may not be immediately recognizable in technology just as they were not quite understood in the new technologies of the past.

The Makers

Visionaries turn ideas into realities by either cooperating with makers, or they become the makers themselves.

When Elon Musk envisioned PayPal, an online system for processing, receiving and sending funds anywhere in the world, he was also the maker of his own vision.

But in the Apple, Inc scenario, Steve Jobs had to cooperate with Wozniak to turn his vision into a reality, without Wozniak it would have been a lot harder for Jobs to bring Apple computers into what they are today.

Makers are not just people who have a skill, they are a lot more than that. they are people that look at the current technology in hand and think about how create something better out of it.

Makers are hackers, they don't settle for the status quo, they are exceedingly good at what they do that they can turn a vision into a reality even if the current technology says it can't be implemented.

And while it's inspiring to learn courage and risk-taking from the visionaries, the makers inspire us to get better at what we do, not just to get the job done, but to make the impossible, possible. They inspire us to go beyond the textbook, beyond what the manual says – to hack their tools and never stop believing that they (and we) can get more out of it.

Some of the most famous makers out there are people like Wozniak, James Gosling, Dennis Richie, Larry Page and Sergey Brin.

One of the most common things among these makers is their devotion and staying true to their values and skills. They reached the top at what they did and turned it into something that changed the world forever.

The Community

The early-adapters, people who are not scared to try the beta or the alpha version of some technology and not just stop there, but to communicate with the makers and the visionaries to help bring their product to reality, into something that could hit mass markets and main stream.

Community could be authors who try the technology and document it, create tutorials, videos or write books about it.

It could also be the contributors in open-source projects. These are the semi-makers who pull technology into their lives by keeping it alive, pushing new features and fixing issues. These are those who are touched by the vision, inspired by the products and are willing to push it further with whatever skill-sets they may have.

Technology advocates also fall under the umbrella of community, except that advocates are a lot more organized and systematic than the average open-source community contributor.

One of the most vibrant communities out there are the Linux communities, with all of their distributions, apps, and kernel developments.

There was a time in history when Linux was frowned upon by corporations which created proprietary alternatives at the time. If it wasn't for the community surrounding Linux, it wouldn't have become what it is today and community keeps pushing Linux, further, morphing it into more cloud-based technologies such as docker containers and web servers which forced even the proprietary software companies to recognize their existence and cooperate even more with the communities to find some business value in their product.

A visionary and a maker who don't know how to create a community around their products can't go very far. The social aspect of technology is of utmost importance and without it, the vision can just be gone, forgotten, or discarded.

Devotion is the most inspiring aspect of communities, the social nature of getting to work with others who share the same passion and communicate with them on that special social sphere.

Communities carry the vision of the visionaries, and develop the skill-sets of the makers, and in some cases, communities will give birth to new makers and visionaries and keep the idea going long after the visionary and the maker are gone or moved on to other ideas.

The Industry

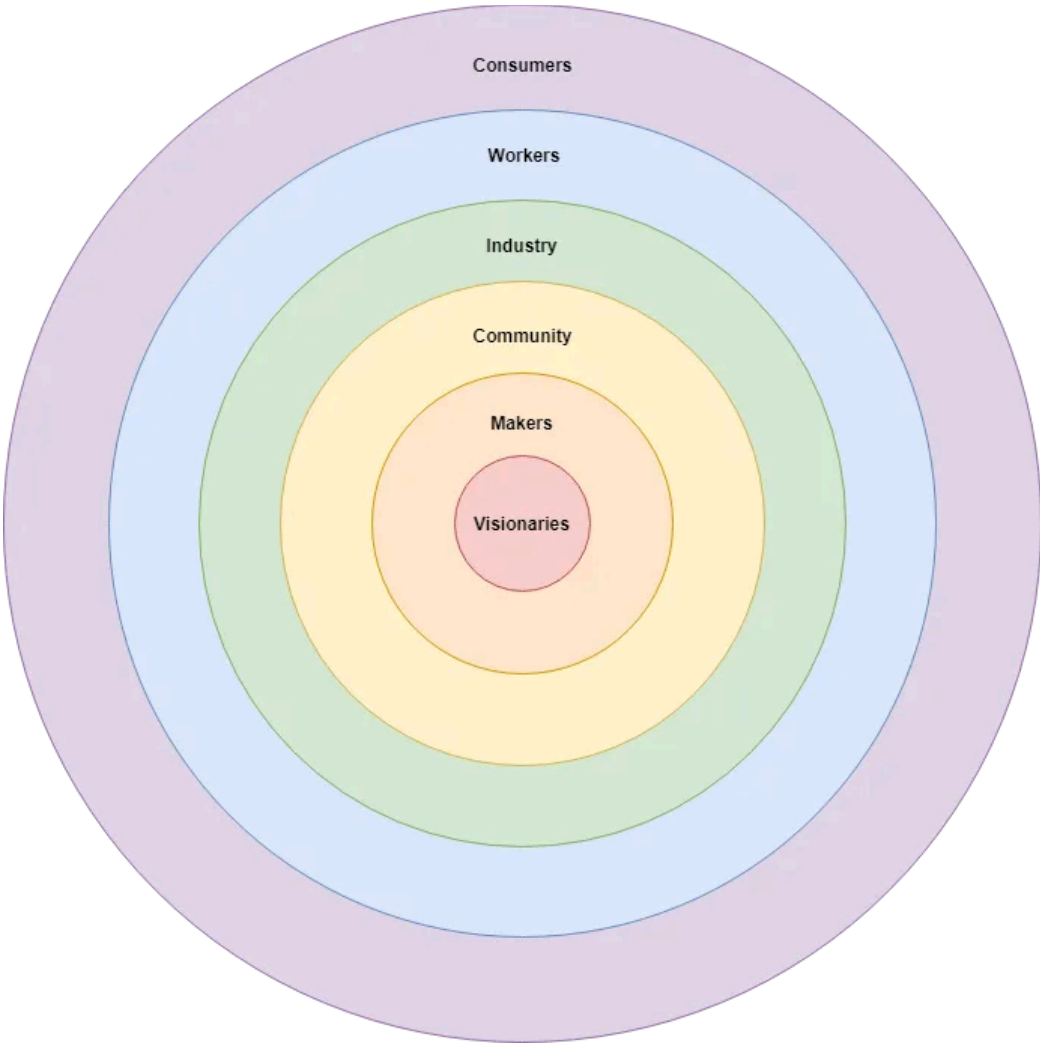
Corporate, startups and even free-lancers are entities that are influenced by the popular, most modern technologies promoted by the communities.

For instance, a few years ago JavaScript was nothing but a simple scripting language that web developers used for some validations on their websites.

But then the community adopted the language and started pushing it to the server-side, built hundreds of thousands of libraries to support every aspect of web development needs, which eventually caught up to the software industry.

Corporations had no option but to rewrite a lot of their technology stack in JavaScript, simply because that was the easiest way to find skill sets in the software market today. It's also the easiest way to find solutions for its most common issues on community powered forums such as StackOverFlow and GitHub.

The adaptation of the industry gives any technology at least two decades of a lifetime before the next new technology takes over.



The Workers

Industries produce their own community of technology maintainers and users, but they are not usually as passionate as the community that made the technology popular in the first place.

When a company adopts a certain technology, new employees will need to learn that technology and use it whether they are passionate about it or not.

And this is when a technology faces it's most important challenge, being used by people who don't really care much for it, and so easy to give up on it if it doesn't get the job done.

Most of corporate workers won't try to go to an open-source project to try to fix a problem with which they are dealing in regards to a certain technology. They would rather just complain and go find an alternative.

Complaining about some drawbacks in a certain technology is a lot better than saying nothing because when a worker who has to use the technology leaves a comment on a community website or a forum, the early-adapters and passionate members of the community have the opportunity to address that and make sure the next release addresses the issues. They can address the issues the workers find in the technology and offer a fix as fast as they can.

This back and forth between non-passionate technology users and the community can either contribute to the empowerment of the technology, making it better, fast and reliable.

Or it can backfire. When the amount of complaints not being addressed exceeds the amount of the contributors in the community to address them, the "hacks" begin to bend the technology backwards, misinformation starts to spread on social media and the technology starts getting "sick" and eventually dies.

Technologies can only live as long as there are maintainers and contributors that are willing to put in the hours and keep the technology relevant to the day and age it's in.

That's why some corporations will adopt the maintainers of the technology and foster communities of passionate contributors to keep the technology alive. But even when someone has a day job to maintain a technology, it's still not as powerful unless they have an inner passion to keep it alive. Technologies may stay on life-support for just as long as a corporation is willing to spend the money to keep it alive, but once it pulls the plug and cuts the funding, the technology ceases to exist.

The Consumers

The last portion here are the consumers. In the software development world, consumers don't really know the specific technology that was used to create a website or mobile app, but they will definitely not support a product that simply doesn't work or requires a lot of time and effort to get something done.

Engineers will question the design and vision of the product just long enough before realizing that the technology itself is the problem and that they need to move on.

End-users are the backbone that keeps corporations and their workers interested in creating products in certain technologies. There have been scenarios in which consumers of products, like Apple's Newton, quickly decided the product didn't live up to the expectations and Apple ended up shutting the project down before it became an historical prodigy like so many of Apple's products.

Knowing the life-cycle of technology helps us predict where it's headed and whether it's worth the investment of time and effort or not.

Understanding how technology is born, grown and how it dies helps technologists adjust their visions, and change their direction to stay relevant, useful and innovative and to continue to provide value to the world.

CHAPTER 19

The Perfect Software

Don't ever think there's such a thing as perfect software, there's no such a thing.

There are, however, relatively satisfying experiences that slowly decline as technology advances and new tools are implemented for better experiences.

The trick here is to build your software to be adaptive to the inevitably upcoming changes before you know what they really are, or how they are going to be.

Because the software you build is nothing but a reflection of who you are, how you felt, and your pattern of thought at the time of developing that software, I see building adaptive software as more than just an intelligent move to build a better experience. It's also a humble gesture that indicates that what you're building is not perfect, because you're not perfect either. Your software is ready to learn new tricks and evolve and morph itself to provide better experiences to its users.

Everywhere in every possible software house you can think of, you'll hear some developers talking about making the code "perfect" – it's like mirage developers endlessly chasing what they'll never find.

This eventually causes a lot of tension between the business and the developers involved, because the business will never understand why they need to spend more money to get the same experience, and developers will never be able to explain to non-technical stakeholders why putting in four more weeks of development without any observable results is important for the business.

Some thinkers (such as uncle Bob) suggest that making the software "right" is a part of the risk developers should take and include that in their development efforts, as a part of delivering features.

But then again, he'll say even after taking that risk, the software itself must be in a continuous maintenance state, because software rots, and it requires continuous effort to keep it up to date.

So, what is an adaptive software, what are the measures of success for an adaptive software?

Modularization

The first and most important sign of an adaptive software is modularization or componentization.

Modularized software with pluggable, configurable and reusable components makes it a lot easier to change these components and adapt new patterns to provide better experiences.

But with modularization comes abstraction, and abstraction is the tough part. Because abstraction isn't just about building contracts that both the provider and the consumer should honor, it's also the art of predicting upcoming changes that you don't fully understand, or know the nature of.

Building contracts is just like predicting the future. It's about how long it can hold off, and both sides try to bend it backwards before they decide to sunset your work and build v2.0.

Building v2.0 is a declaration that your prediction of the business needs only lived so long before you went into this dark area where all your predictions and expectations have failed.

That's why I stress on the notion that developers and engineers are two very different creatures. The first thinks all the problems lie in the code in front of their eyes, while the latter don't just think of the big picture and the entire process end to end, but they also consider how the software will actually hold up for the next three to five years before it turns into a legacy that everyone tries to stay away from because it's a pain to understand let alone support.

The Experience

In addition to modularization, there's the other side of the coin, which is the experience.

A successfully adaptive software should evolve not just on the back-end but also to the end-users. They should see something better, not just the same experience, but better, faster, easier and feel improved intuitiveness.

In Steve Jobs' own words, he used to say: "Our software should know what users want before they even know they want it."

And as hard as that is, it's actually a very achievable, realistic experience that Apple succeeded at some point in time and turned it into reality.

Using the latest UI framework isn't enough to build a better experience and provide an actual business value to the end-user, just the same way buying the fastest car doesn't magically turn you overnight into an F-1 racer.

Modern tools will help you turn your vision into a reality faster, but if you have no vision and your software isn't experience-driven, then the tools are just as good as any others. Tools don't inspire vision, vision comes from the tools you were born with - your mind, spirit and personal experiences.

David Pullara

While I was writing this article, my friend and mentor David Pullara brought a good point. I can summarize it in one excellent question which is, "Are all modern adaptations of software good?"

And my answer is that I don't really think so. Some adaptations might cause an end to the software forever and a loss of its customer base.

Engineers must be intelligent enough to understand that building adaptive software then evolving it must be governed by some principles whether the adaptation will actually contribute to both developers' and users' experiences or not.

I've been to companies where they decided to adapt a new technology just because it's new (and shiny) – but, that adaptation brought a miserable experience to both developers and end-users.

The Future

Engineering software that will be used by users that have different tastes, opinions and accessibilities can be very challenging, especially considering that these very same users could have two different opinions based on their moods, what devices they used to utilize the software and all different levels of variables.

A deeper understanding of people in general, staying in tune with how different people react to different software experiences and interacting and communicating with a wide variety of people can immensely help build better software that people could actually appreciate.

Dear engineers, don't lock yourself in your office building and design software for users you haven't even met once. Get out of your offices and meet with your end users, listen carefully and welcome their feedback, challenge their thoughts and think deeply about not just what they are saying, but what they are *really* saying.

In some companies there are product owners and program managers to do that part, but then a lot of things get lost in the translation. I highly recommend that even if you have a product owner on your team, try to get involved in these end-user meeting. Sit and listen, take notes and put yourself in your potential end-user shoes. This is how you build software.

My favorite story here is when Steve Jobs wanted to sell a printer to potential customers. He didn't really show them the printer, he showed them a piece of paper that had the print on it. He showed them what the printer could do. He sold them on the experience. It's the end product that actually matters. There's a lot to learn there.

CHAPTER 20

Chickens, Half-Ways, Almost-Theres and Winners

I've come across this interesting quote recently by Stephen King where he says:

"Talent is cheaper than table salt. What separates the talented individual from the successful one is a lot of hard work"

And this is where I started thinking about the very concept of "hard work".

I started asking myself how much hard work is hard enough? How do we quantify that? What are good examples and bad examples to learn from about hard work?

In the software engineering industry and so many others I'd assume, people fall into one of four different categories:

Chickens

Chickens are the early quitters; they are too scared to even give something a shot, or cannot be patient enough for a good amount of time to see if they fit into something or not.

Chickens have a very interesting mindset, somehow. They know they won't fit into something without even trying, and if they did try it, they do it half-heartedly with no true intention whatsoever to see things through.

And these folks spend most of their adult lives working low paying jobs they hate, hoping the perfect opportunity will simply reveal itself to them.

Chicken people are those who live under the impression that the world somehow owes them something. They expect from the world more than they give. They play no role in anyone's life and they have no impact whatsoever on the future of humanity.

It takes some effort to find what we are good at. It doesn't always just happen that you know your purpose in the world. It takes a lot of trying, courage and patience, all of which the chicken type of people obviously don't have.

If you've ever been a chicken type of person, or you still are, here's what you can do about it:

Give things a shot long enough, maybe a year or so, and do your very best to be the best at it. Stay at something even when it starts to feel uncomfortable because that is because if you live your whole life not trying things, you will end up being a person with no purpose, no meaning, and no impact in our world, Is that who you want to be?

The Half-Ways

The half-ways are a bit better than the chickens, but they are not much better than anything else.

Half-ways are the people who had the courage to get something done, went half-way through a project but quit half way, because it was "too challenging" or "too hard" or whatever other excuse came along.

Half-ways love to blame everyone else for their problems and they expect everyone to help them. They don't even try to give something enough effort before seeking help from others, and most of the time the minimum amount of effort is all it takes to figure something out. But these are the half-ways we are talking about, so they won't even give things a solid try.

Being a half-way means you're a half-baked cake, no one wants to buy that or eat it even if it was for free!

If you're a half-way person, congratulations! You've had the courage to try something new! But you've come short at the resilience part. Don't break in front of the difficulties you may face doing your job. Fight back! Then fight back harder until you win it. If it was easy, everyone else would've done it, then how special would it be?

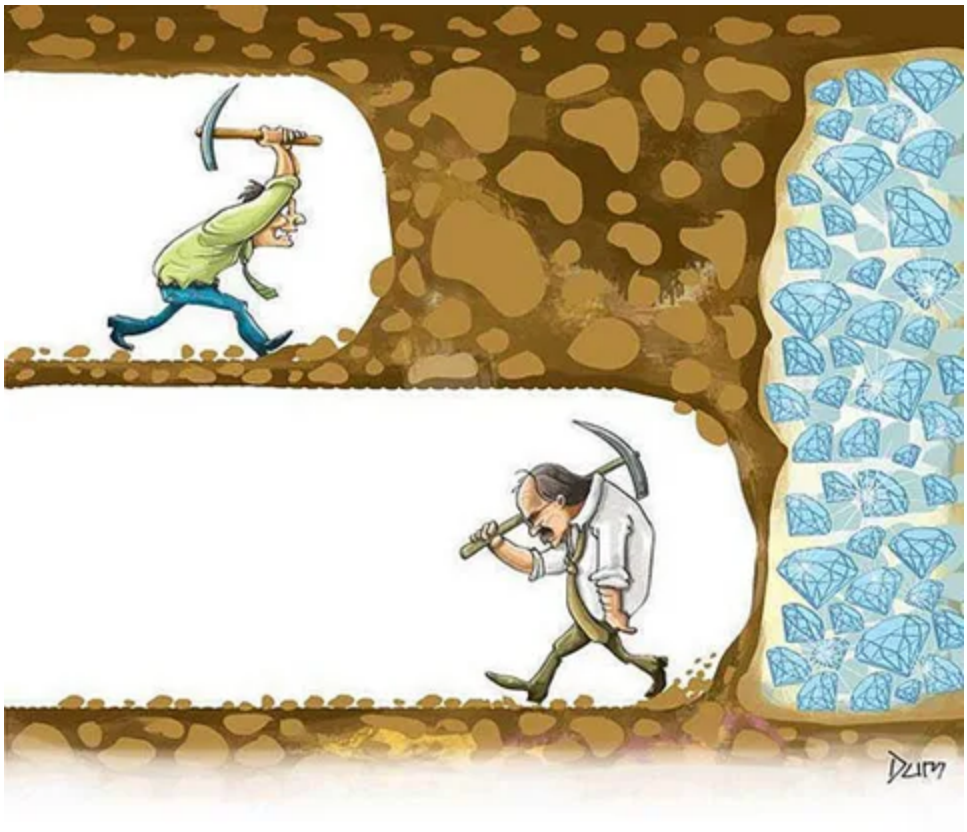
The Almost-Theres

The almost-theres are those who quit right before the job is done.

It's not always clear when the job will be done because some projects keep going on and on with no vision in the near future for it to be done.

This can also apply to certain problems. Some people quit before trying all kinds of different solutions to solve a problem when there's only a couple of possible solutions left!

And those folks have done "almost" the hard part, (the part that the chickens and the half-ways couldn't do, but they fell short on the persistence part) but they fell short after they struggled with a lot of problems and issues.



“Almost-theres” are the ones who pay the highest price – take the biggest toll - amongst all the others we have discussed, but unfortunately, they get back in return almost the exact same results as the chickens and the half-ways.

Persistence is the core value that distinguishes a hard-worker from a quitter or a loser.

Only those who persist in pursuing an accomplishment are the ones worthy of winning, because most of the time persistence is the only fuel you need to get to the places no one ever reached before.

If you are an “almost-theres” person, my advice to you is to fuel up on persistence, tell yourself that you’re almost done with the project, because most of the time you’ll be absolutely right. Part of every journey is the end, nothing goes on forever, so stand your ground and be strong.

The Winners

The winners are those who take humanity to the next level, no matter where they are, who they are and what they are.

The winners have gathered the three core values of success which the previous three have lacked: patience, resilience and persistence.

And they have earned their success rightfully so.

Because winners are those who spent the nights working while others were asleep.

And winners are those who put in the hours when others decided to play.

And winners are those who are courageous enough to try everything, question everything, examine everything, and not take anything for granted. They leave no rock unturned and no question unasked.

And winners are our hope for a brighter future and a better world. Without them we're lost, and without them nothing would have changed for hundreds of years.

Everyone is born a winner. Look at the children learning how to walk. They try and fall, and try and fall, then try again and fall. Then they get it right and walk. Then they run and jump and play!

But somehow along the way some of us form a loser mindset. We desert the gift of resilience, patience and persistence we were born with, and we spend the rest of our lives in that state.

Now I'm going to share one of my favorite quotes about hard work from Conan O'Brien:

"Nobody in life gets exactly what they thought they were going to get. But if you work really hard and you're kind, amazing things will happen."

CHAPTER 21

Failure is a Mindset

The binary nature of the software industry reveals more clearly those who are set to achieve their goals and those who don't have any goals at all.

As I work on a daily basis with all different kinds of engineers, it becomes clearer and clearer to me two very interesting mindsets that almost overpowers most of the engineers in this industry.

These mindsets are the failure mindset, and the learner mindset.

But learning in and of itself isn't enough if it's not combined with the passion and zeal that triggers more attempts to either learn more, or finally achieve the goal.

In the software business things either work or don't and that makes it very clear whether the design and development and all the efforts involved were fruitful or not.

Never in the history of software businesses has a client agreed to pay money for a non-working software, and the less-techy clients are, they seem to be more focused on the goal, more than some of the developers who are making the product themselves.

Let's talk about the failure mindset.

There are symptoms that clearly shows an engineer who's plagued with a failure mindset from an engineer who doesn't.

Blame Game

A failure mindset likes to blame everyone around for not achieving their goals. They'll blame the technology they are working with, or the people who invented the technology, or the team they are working with, or anything, or anyone they can find to blame for not achieving their goals.

But a person with a failure mindset never blames themselves for not trying again – trying harder and differently.

Victimizing themselves. People with a failure mindset never try to overcome the obstacles they are facing because blaming it on someone else (even if it was rightful) is the easier route to not having to handle the task at hand.

But on the other hand, it causes not just a delay on delivering a product, and not just a cost of time and effort and funds on the project, but it has an even deeper impact on the team and the overall morale for everyone.

I say it has a negative impact on the team because a failure mindset wouldn't be a source of inspiration and perseverance for others. In fact, it sometimes infects others to settle for failure and find excuses to blame the world for that failure, but themselves of course.

This mindset that shows a person trying to lean and grow, not afraid to fail and try again and again – who doesn't really care what people say and who surely doesn't try to hang it's errors on others, and takes full ownership and responsibility for mistakes and has the courage to stand up and correct the errors.

Purposelessness

A failure mindset has no purpose, no goals, nothing to chase and no dreams to catch.

People with failure mindsets live a whole life, maybe 60, 70 or 80 years and they have impacted no one's life, including their own.

They have contributed nothing to world other than providing manure to fertilize the land, and even then, animals seem to do a better job at manuring the land than those with failure mindsets.

Because they had no goals or purpose to achieve and even if they did, they didn't try to make it happen. It was easier for them to simply say: life is hard and settle for nothing, and nothing they become.

People who work jobs they hate, are purposeless. They waste their energy complaining instead of trying to change the status quo and work with what they got to create a better future.

When in Rome

A failure mindset lives with no principles. It's one and only goal is leech off of the world, and work whatever job is available rather than trying to build a career.

In the engineering industry, a mindset with no principles will violate as many rules as they can, and cut as many corners as they could, to just be done with the day, and not have to deal with the consequences of not delivering a task.

In this industry it becomes so clear so fast those who properly name their variables, extract their methods and document their code from those who are just employees They don't think of engineering as a craft, but more like a job that pays money and that's all, That's it for them.

I've seen people switching patterns, best practices and technologies just because the job pays more, but not because they think it's the right thing to do. And not because they foresee any growth or learning opportunities in that.

Money is very important, but when one has a purpose, money because a side-effect of healthy learning, growing and achieving mindset rather than being a purpose in and of itself.

Lack of Passion

A failure mindset produces a lack of passion. Passion is what triggers the mind to try again, and do more and think differently to achieve its goals.

You can see the lack of passion in many simple actions, such as a general indifference, showing up late to every meeting, giving half answers to every question and having no opinion on any matter.

Spending as little time as they can to accomplish a task, and working just hard enough not to get fired. Because not getting fired is their end goal, just keep paying me, they would say.

Showing up late and leaving early, no heart in what's being developed and no thought in what's being designed, it's just a job, they would say.

Unhappiness

A failure mindset knows it's not living up to it's full potential. And that's why it's unhappy, sad and depressed.

They bring that negative energy to everyone working with them, downplay others achievements and see the negative side of every attempt to achieve more and do more.

Bob Ross said: "You do your best work when you do a job that makes you happy"

But how can one find a job that makes them happy if they are not trying?

Let's talk about the learner mindset!

A learner mindset could come from a failure mindset, we've all been in a place where we lost hope, passion and were confused by what we need to do next, trying to find our place in the world.

But changing a failure mindset requires a strong will, a solid intention and an actionable plan to change one's life.

It all starts with finding one their purpose, what they were meant to do. And that requires a trial and error with as many careers as possible out there.

I've seen people do the oddest of jobs with passion. They earn and help the poor and make the best of what they have. They do what they do so well that people think it's easy but it's really not.

A learner mindset is a trial and error junkie. It never quits, surrenders and gives in to the status quo.

It inspires and teams with others and brings life and happiness to those with whom they work.

It sees the bright side of every attempt, and wishes for the best in every trial and sees the positive in every error and learns from it.

It encourages others to do the same, tries to open their eyes to see the horizon of achievement and success, tries to be the best for the team, not the best on the team.

A leaner mindset doesn't just get the job done, it gets it done right, with passion, and heart and hope to do even better, even when the world is happy with what's already achieved.

Because a learner mindset never stops believing that it can get more out of something, learn more, make it more, make it the best, and the best of the very best!

People with learner mindsets are the people who changed our world from what it was thousands of years ago to what it is today. They are our hope to find the best cures for our most stubborn diseases, prevent our worst disasters and inspire our wildest dreams.

They take us to the moon! They get us to our beloved ones faster. They connect us with each other and help us be healthier smarter and more hopeful for a better tomorrow.

They look at the impossible and see … I'm possible!

CHAPTER 22

Pain & Passion

If you're not driven by passion, you are driven by pain.

That's just how everyone is. in their day to day lives. Some people wake up in the morning to go chase their dreams, change the world and leave their mark in history.

And some others wake up only to pay their bills, work just hard enough not to get fired and make sure they don't end up being homeless when they retire - it's just another day for them.

Where do you stand?

Heart & Soul

The difference between the two types of people is more than just obvious, especially in an industry like software engineering, you can see so quickly those who cut corners to move cards across the board and deliver "features".

And those who take the time to put their heart and soul into their work, try to best themselves, learn of better ways and smarter methods to accomplish faster, it's in their devotion to quality in every line of code they type and every design they envision.

Which one of those are you?

Titles, Salaries, Benefits and Credit

People with passion don't really care about titles, salaries, and benefits. Because all these things are side effects to success, but they are not a success in and of themselves.

Instead, they chase success, they chase changing the world for the better, making people's lives easier, regardless of whether they get credit or not, their true reward is to see their creation helping others, and making a difference in the world.

But people with pain will fight tooth and nail over titles and salaries. they will step over each other, lose their relationships with their colleagues, gossip and take credit so they can get that promotion. Because the promotions and salaries are the only validation they could get.

Their work is below average, they change no one's life, and their end products live shorter than their careers and soon turned into a legacy product that no one cares to maintain.

On which side do you stand?

Yes Man!

People who are driven by passion are the furthest from being a yes man. They will disagree, and work hard to prove their point. They know how and when to say no, because they have an opinion about something, they are driven by a passion and a vision that makes them see very clearly what a better product would look like, it's the quality that drives their efforts not appeasing their bosses.

Because people with passion believe in something, they fight for it, they may win, and they may lose but at least they put up a fight to deliver a better quality and a useful product, so it's totally worth it for them.

But people with pain will agree to whatever the boss says, they are the Yes men of the world, the product's quality is the last thing to think about as long as their leaders are appeased and happy.

Because their end goal is to be validated by their bosses not by the quality of their creation, the product of their minds!

Are you a Yes Man?

The Smile tells it All

People with passion have a glowing smile every day, because being driven by passion brings in happiness, because everyday means to them a step closer to achieving their goals!

These are the ones who will put in the extra unpaid hours to make sure things are done right, they will spend out of their own pockets to buy books, go to seminars and attend conferences to learn more, achieve more and be more!

People with pain seldom do they smile, everyday is just a struggle, a fight with life, everyday is just another bill to pay, no meaning or purpose, they cycle is eat, sleep, work, pay bills then repeat!

You can tell a person driven by pain on a Monday morning, the frown and the attitude tells it all, a Monday to a person with pain is torture.

But a Monday to a person with passion means a new opportunity to achieve more! Because people with passion will never stop believing they can get more out of their day, let be a Monday or a Saturday, it doesn't matter what day it is, they have a dream and they are set on achieving it, whatever it takes.

Do you feel excited to go back to work on Mondays?

Mistakes

People with pain deny their mistakes, they play the blame game, their self-image is more important to them than their own growth.

They seldom learn from their mistakes, but rather learn how to not take responsibility for them. They see mistakes as a threat not as an opportunity

But people with passion know you can either win or learn. They know mistakes is just a normal part of the process of achieving one their goals, they know if they are not making mistakes, they are probably not trying hard enough.

And because they see mistakes as a part of the process, they see no problem taking full responsibility for their mistakes, owning up to them and learning from them and sharing their knowledge to make sure no one would repeat the same mistakes.

These are the ones who push everyone further, they push us toward a better world, they work with a collective cohesive mindset.

Do you fear admitting your mistakes?

Spread the Light!

People with passion want to help others succeed, because they know when they work with others they can get further faster.

They are always the ones lending a helping hand and inspiring others to chase their dreams, they are the ones always volunteering to share to knowledge, spreading that passion wherever they go and however they can, because they know how beautiful it tastes to be passionate about something, believing in something and accomplishing something!

But people with pain are self-centered, selfish creatures.

They only care about their self-interest, they are always worried someone is going to come in and take their "jobs".

These are the ones who worry about "job security" – they have no interest in sharing what they learned, they will make sure their work is so complex and mysterious no one could understand it or maintain so they will always be needed.

These are the ones who stick to the same project for half of their lives, they never move on to creating more products, they are a cog in a machine of their own making, they are just as dull as a hardware component, there's no purpose there, they can only be purposed by others, but they have no purpose on their own.

They see every attempt for modernization or change as a threat, because it means they need to learn new things and adapt to new ways, people with pain don't want to learn new things, they want to come everyday and do the same thing because they have no intention of changing anything.

Do you see change as a threat?

Finding Passion

Finding one their passion is a tough job; it may take years for some to find what they're passionate about. It takes courage and determination to continue to try different things until one finds their purpose.

We were all created for a purpose, and we were all created with brains to find ways to know what that purpose is.

Some of us find it so early, and they just know this is what they were meant to do.

Some others try a couple of things, then they give up, they say I have no purpose, that's the ultimate defeat.

But some others will keep trying to find their purpose until their last breath, just like this **81-year-old woman** who learned programming and created her first iPhone app.

The thing that fascinates me about Masako Wakamiya is that she was born at a time and age where mobile apps weren't event existent.

She tried everything her whole life, until mobile apps came to light, and that's when she found her purpose, she didn't quit even when she has become 81!

But finding one their passion is one thing, acting on it is a completely different thing.

Everyone in the world at some point in time have had an idea that may change the world, but only few decided to take the risk and act on it, going down the rabbit hole and seeing how far it goes.

These are the people who created most of the most popular products that we use today, from inventing electricity to self-driving cars, from mainframes to wearable technologies that fit in a finger ring!

Mary Oliver says: "The most regretful people on earth are those who felt the call to creative work, who felt their own creative power restive and uprising, and gave to it neither power nor time"

Did you find your passion yet?

What are you going to do about it?

CHAPTER 23

Innovation Engineered

Where do innovative ideas come from?

What does it take for one to find their genuine idea and change the world?

When we talk about innovators, we talk about a very special group of people who do not just conform to the status queue, a group of people who do not want to just live like everybody else.

A group of people that realized that those who created every other innovative thing around them aren't better or smarter than them in any way shape or form.

And that realization is what inspires them to become innovators themselves – seeking to leave their mark in this world and make a bold statement that they were here, and that they had so much passion and love for this world that they wanted to make it better, even long after they were gone.

When we talk about innovation, we talk about ideas that turned into realities that shifted the way the world thinks, operates and in general solves it's problems.

But innovation isn't an easy goal to achieve, it requires a ton of effort and struggle to open it's doors to someone so they can change.

So, the obvious question here would be, where do innovative ideas come from?

I have been studying innovators closely reading about their histories and lives in the past few years and I gathered few similarities between them, let's discuss those in detail:

Looking Beyond

Innovators don't seem to conform to anything without a convincing proof or evidence.

They question everything, challenge everything and fear nothing.

In the dark ages, innovators faced a lot of persecution for their ideas, their books were burned their houses were damaged and their families were separated.

But that didn't stop them from moving forward for what they believed was right, they believed that if you were born with such imaginative brains, then it's a birth-right for everyone to logically examine their surroundings, and experiment with other assumptions then observe the results.

As an engineer when I read these stories, I can't help it but see a pattern here.

A pattern of assuming that the foundation of every approach even though successful might not be necessarily the best foundation for what the approach was trying to solve.

Let me simplify this further.

If you walk into a room and you need to change the lightbulb, then you see everyone around you using chairs to reach the lightbulbs to replace them.

An innovator would think, are chairs the best option to solve this problem? What if the lightbulbs were further up to the ceilings? A semi-innovator would say: "let's build higher chairs!"

But an innovator would say: "Maybe the chairs aren't the best option for this problem?"

They challenge the foundation of the assumption, the current tool, they examine better options further down, because they understand that sometimes a foundational assumption could be the cause of a further unnecessary restriction that can be fully avoided once the foundation is changed.

But it takes courage to challenge the status que, because in a lot of societies challenging the status que could mean being disowned, disserted or disrespected – it can even lead to persecution and death.

It takes a courageous heart and a deep convention mixed with strong passion to ignore all of that and move forward with one's ideas.

Innovation always flourish when the restriction of free thinking decreases in the society, however, there are no society out there today that doesn't have some restrictions in some form or another, the job of the innovator is to find the best society that is lenient with it's rules against on idea versus another.

Emotional Connection

An idea isn't fully thought through if there was no emotional connection attached to it.

Plato says: "Necessity is the mother of invention. A need or problem encourages creative efforts to meet the need or solve the problem."

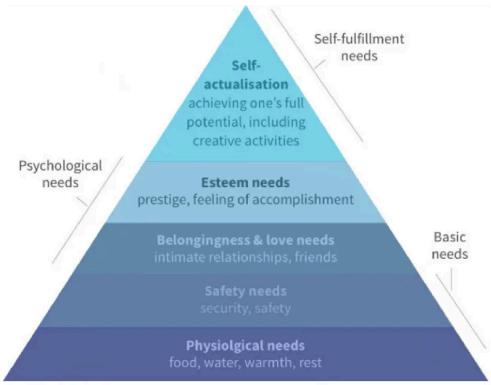
In other words, when you're looking for your next innovative idea, it needs to be something that bothers you personally, something that you deal with on day to day basis and decide you can make it better!

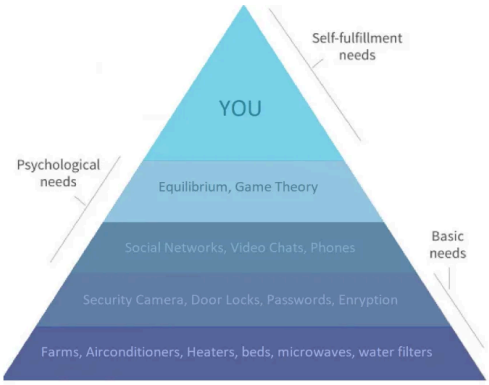
Looking at history, there are a tons of example of innovators that invented amazing things out of necessity, such as Joseph-Armand Bombardier who invented the snowmobile after his son Yvon died and couldn't take him to the nearest hospital because the roads were snowed in and there was no means of transportation through the snow back then.

And so is the case with so many other inventions, such as wheelchairs, gas masks, polio vaccines and so many others that were born out of necessity and a deep emotional connection to the problem.

But if you look around, there are other inventions that didn't have the traumatizing form of necessity – it had a different form of necessity – social recognition.

Looking back at Maslow hierarchy of needs, you can easily see how much of the needs were the biggest drive for innovation, I mapped Maslow's Hierarchy to some of the biggest inventions to show you how things played out through our history of innovation:





There are several forms of necessities that drive innovation, it's not always a life or death situation – there are many drivers for innovation, Maslow's hierarchy is so genius that I couldn't think of anything out of these needs he presented.

Perseverance

When I look at the history of most of the innovators that shaped our world to what it is today I can see clearly they had so much persistence and resilience, they were trial and error addicts – the many thousands of failing experimentations they made didn't deter them away from continuing to try.

Edison once said: "genius is 1% inspiration and 99% perspiration."

Edison is one of the greatest examples of perseverance – he tried over 1,000 times before he was able to invent the lightbulb, and when a reporter asked him "How did it feel to fail 1,000 times?" Edison replied, "I didn't fail 1,000 times. The light bulb was an invention with 1,000 steps."

Edison is one of the greatest examples because his perseverance led to over 1,093 patents. And while many of these inventions — such as the light bulb, stock printer, phonograph and alkaline battery — were groundbreaking, even more of them were unsuccessful.

So accepting failure as a step not as a setback was a mindset that led to innovation in Edison's case.

But history is full of examples of people like Edison who have the same mindset.

And when you realize that everyone in this world is born with at least one genuine idea that could change the world as we know it, you can see how crucial perseverance is to the process of innovation considering how many decided to quit on their dreams and become just like everybody else … a nobody!

As an engineer, when I hear the stories about perseverance, I hear the mathematical term permutations, different combination of things and the ability to keep going forward until all permutations have been examined.

Our world is so complex and infinite that it has infinite number of permutations and possibilities, it's a lot easier for engineers today as technology have advanced to model a problem and run all possible scenarios and simulations in a lot shorter time than what it used to take back in the old days when everything was manual and analogues.

Modeling and simulation through computer programming and software engineering has opened the doors for innovators to experiment and try multiple things in a split of a second – which means the innovation process has been expedited to a maximum to get our ideas examined a lot faster than before.

You are an innovator too!

Everyone in this world is dealing with a problem, some fall into line and do what everyone else is doing, sometimes by doing absolutely nothing, and some others decide to do something about their problems.

And because everyone is dealing in one way or another with a form of a problem in their lives, regardless of their social status, monthly income or where they live, that also means they are capable of solving these problems, capable of finding the right idea and the best innovative solution to solve these problems.

You owe it to yourself to find your genuine idea and solve your problems intelligently and change the world for the better.

You owe it to yourself because you're capable of innovating, regardless of who you are, where you from and what your education or daily job is.

Don't go quietly into that good night, living like everybody else, being just a number that has no meaning or value or cause to live for.

Look around you and find a wrong that you need to make right, an evil that you need to turn good, listen closely to what your heart tells you not what people say, you are unique and special in your own way, and you have the capabilities and the energy to make your uniqueness a reality.

“Do not go gentle into that good night,

Old age should burn and rave at close of day;

Rage, rage against the dying of the light.”

— Dylan Thomas

CHAPTER 24

Entrepreneurial Innovative Engineering

A software engineer that has an entrepreneurial sense is ten times more likely to build a market-ready software than an engineer who doesn't.

That's simply because any commercial software out there requires more than just good engineering practices, flawless architecture or resilient infrastructure.

A successful commercial software has more to it because we as end users for that software are very complex in our nature, we agree on things and we disagree on others, sometimes we agree on things during certain times or seasons and we completely disagree on the same thing on a different time or occasion.

A successful software that is supposed to be used by people, with all their differences, tastes and perceptions requires an entrepreneurial sense to know when to enter the market, what risks can be taken and what risks needs to be avoided.

Practically speaking this translates into some practices and principles, a mindset that knows where the energy should go and where time shouldn't be wasted.

Being an entrepreneur and an engineer myself I have learned few things being in business for so long that I'd like to share with you in this article.

History of Entrepreneurial Engineers

Let's look at most of the successful software companies today, Jeff Bezos, Bill Gates, Zuckerberg and Musk, these were engineers that developed a sense of innovation and evolved entrepreneurially to deliver their innovative ideas to the world.

It's interesting to see how much fraction and downfalls Apple have had to go through running with just a business guy like Jobs, how much he relied on and conflicted with engineers, but you don't really hear much of these stories with Microsoft, Facebook, Google or Amazon – at least not at the scale of Apple.

Apple is a good example of how much struggle an entrepreneur has to go through if they don't have the technical background, Jobs was definitely an innovator, he had a sense of elegance and style and he wanted to make technology an extension of the human person – but his lack of technical knowledge caused him a lot of miscommunications between himself and the engineers he worked with, it became very obvious with the departure of his top engineer and best friend Wozniak of Apple in 1985.

Learning from history and mixing that with personal experience I've started seeing a pattern there – questions that I've always asked myself about what it takes to make one software company so successful that it's budget tops an entire budget of a whole 3rd world country versus another that did everything "right" but didn't get as far?

Let's talk about these patterns ...

Getting There First

I cannot tell you how many times I was able to score a deal with a client or land a job in a company by simply being there first, being ready and on time.

This is the least thing one needs to do and it requires almost no skillset other than being alarmed and cognizant of time and paying attention to detail.

The very house that I live in these days, I was able to land a great deal on it simply by responding to the owner within 10 minutes of them texting me, he said to me I am going to sign a deal with you just because you're ready and you're on time!

I didn't think such a thing would give me much advantage, but evidently, it gives you a great advantage to be the first and the fastest but it's not always a guarantee, but it's definitely a factor!

Because with the rhythm of life and technology today everything is happening so fast and people aren't as patient as they used to be, that's how the market looks like today, a good engineer needs to adjust to that.

When Jeff Bezos from Amazon was interviewing engineers one of the questions he would ask was: Can you get the job done in 3rd of the time a regular engineer could?

There's a reason for that, being first gives you advantage in the market, makes you more familiar, more common, and it's hard for a lot of people to adjust to new technologies when they already have something that does the same thing, so being there first definitely makes a difference.

And big software companies understand that, you can see how many of these companies started developing VRs and Cryptocurrency – Gaming and collaboration tooling and even in the mobile world how many started to explore the foldable phones options even if it's not fully market-ready like what happened with Samsung.

But being there first, and not being market-ready is a death march, if you don't have an already established reputation and a user base, it could mean that the reputation you non-market-ready product will illustrate the reputation of your software company forever, you'll be known as the company with bad products and this other company does it better!

Take Tesla for example, electric cars existed long before Tesla, Google had self-driving cars but they weren't market-ready, this is how Elon won that market, he said I could do this better, I can create a market-ready product!

Now other companies like Nissan, Mercedes and others are trying to catch up that trend, but people will be more likely to buy a Tesla than a Nissan Leaf for being both there first and being market-ready.

So, what's market readiness?

Market Readiness

Just because your software works, and it looks "pretty" it doesn't mean that it's market ready.

There are apps and websites out there that don't exactly live up to the standard of design and best practices some other draw, but they are more popular, more profitable and more agile and useful!

Craigslist has basically no user interface, if you look at Craigslist today it looks like a website that was developed in 1998 – a bunch of links all over the place, yet they are a one-billion-dollar company today.

So, what's the secret of market readiness?

Market readiness is the ability to build something the simplest user would be able to use and understand.

I have seen 3-year-olds who knew how to use an iPad navigate through apps and pick up which videos the want to watch learning all of that in a couple of months!

Being simple and user friendly makes your software more market ready than others.

But simplicity isn't everything, there's more to market readiness than that.

It's important that your software touches on a recurring need into people's lives, something they feel they need to do on daily basis, like social media!

People feel the urge to communicate and be recognized on daily basis, something the back of their heads pushes them to feel belonging to something, a group, a family or whatever it may be.

If you look at Maslow's hierarchy of needs and compare it to the most successful software out there you will start to see the pattern, some of these patterns are destructive and some others fulfill a basic need that everyone could relate to.

There's a bigger game in the market that people call "creating the need" – creating the need for a product basically touches on the exact same thing as any other product, basic needs, getting something faster with the least amount of effort possible to fulfil a specific need.

Amazon gets you food, toys, furniture and everything else within two days – you type in what you're looking for and you click a button!

Facebook enables you to find like-minded people, groups, family and friends – type in the name and click add friend and you're connected!

Market-readiness relies heavily on recurring needs, they could be daily, bi-weekly or even annually, whenever they maybe – don't create something someone "might" use every "once in a while" that's a product that's not ready for the market.

But market readiness today is more complex than that, it requires targeting a specific group of beta testers, running surveys and examining feedback before going live to the public!

The technology you use must give you that agility to change features and deploy faster and gather information as fast as possible.

Let's talk about the technology then!

The Technical Tools

As a software engineer, there are today hundreds of tools that do exactly the same thing, building websites with no coding experience, building websites with little coding experience, testing mobile apps – content management systems and many, many more for every aspect of every software development platform out there.

Some engineers randomly pick up one of these technologies maybe because this is what everyone else is using or this is what the "cool kids" are doing or whatever other reason, and they tend to build every product and every project with these tools.

But sometimes the tool is a huge time-wasting blocker, that makes the process of pushing new features and developing new products nearly impossible without spending days and days configuring and adjusting.

A part of that could be because of the bad architecture, pattern or development style, but another part is the technology itself.

I've seen some of the most enthusiastic people about certain technology fully hating it as they see how slow it makes them and how crippled it causes their product to be on delivering new feature and fixing defects.

An intelligent engineer picks up the technology that takes care of the redundant stuff, let it be scaffolding an interface for a CRUD operation or building a pipeline to share a project amongst other engineers.

Some engineers run after the newest "shiny" technology just because it's new, they don't give it enough thought to see if it's market-ready – will it help them get from point A (being the idea) to point B (being the product and end users) as fast and as easy as possible.

For years I've been personally using technologies such as C#, Visual Studio and Azure and here's some of the comments I received back from some of my clients:

"Hassan works quickly and is a patient and thorough explainer. He has sent me two demos - narrated videos explaining the product step-by-step. He is very adept at what he does, and his technical understanding is mind-blowing. "

Here's another comment from a different client:

"I am currently working with Hassan on multiple app's and have been pleased with his timeliness and simplicity when developing app's. He takes time to think through the project and deliver a simple solution through his programming skills."

The common theme here is "working quickly" and "timeliness" – if I hadn't used the right technology I wouldn't be able to receive that level of recommendation – some of these changes would take from 5 – 10 mins to be done, and customers are blown away by how fast things can be done, it doesn't mean the feature is simple or easy, it means the technology I'm using is helping me get where I want to go faster.

But communication and understanding are a very important aspect of this, if you cannot communicate with your customers, clients or project managers then all of this would fall part, which is what we are going to talk about in the next article ….

… To be continued

We talked in the last article about picking up the right technology that gets you further, faster!

But technology without communication skills is completely useless.

If you cannot communicate the most complex technical terms to your clients in a fashion they would simply understand then you're just an engineer – someone who's incapable of communicating with others except for computers.

Let's dive deeper into communication.

Communication

As an entrepreneurial engineer myself I cannot tell you how important it is for you to be able to communicate effectively with your clients. It could save you a month worth of development effort just by negotiating what they need from a usability standpoint.

But if I were to describe communication (with humans of course) to an engineer I would say it's the ability to understand the cultural language your client is speaking.

You and your client could be both speaking English to each other, but the common ground between the two of you might almost be non-existent.

Let's take an example of a client who wants to build a website for construction tools and machinery. You need to go learn about construction and watch a couple of YouTube videos to get a glimpse of how construction folks talk, think and what common terms they use.

It's important to do that because if your client doesn't feel that there is a common ground they will lose interest in working with you, they will feel disconnect and unable to communicate their vision to you.

And believe me when I say, whoever your client is, the very fact that you know how to develop software makes them think you're already a genius – they will try to come out as tech savvies who know the ins and outs about software.

Don't stop them or embarrass them when they get it wrong – they are not then technical people in this situation - you are!

And it's your responsibility to educate your clients and take care of their technical needs, make them feel comfortable not knowing particular things and be willing to always inform them for as long as they need about a specific technology or a method you're following.

Communication also means being able to live at least a year in the future ahead of your client – understand what they might ask for in the future versus what they're asking for now.

When you live in the future, ahead of your client, you design your software in a way that'll make it easier to develop potential new features, features that are inevitable if your client is willing to continue the development of software.

There's a common belief in the software industry that software engineers are not good communicators – which is a belief that created a whole new type of jobs like product owners and program managers.

I personally believe that communication just like any other skill (such as programming), requires training and practicing – trials and errors, knowing that you will run into situations where you'll be completely rejected or dismissed and you should be okay with that because people are different and you are not for everyone.

When engineers employ their highly trained brains into improving their communication skills they become more efficient and communicative than any others.

An entrepreneurial engineer understands that a good product is completely useless if there's no way to introduce it to others so they can benefit from it, and in order to introduce a product to somebody you have to have the right skills to communicate with people, to show them how great the product is and what it's capable of doing.

But communication isn't just the words you say, it's when you say them and how you say them.

And while that might sound complicated, in reality it's not that complicated at all. It's just like programming – it might look complex and hard for someone who's not familiar with it, but just like any skill once you practice it, it becomes easier and easier to do.

Communication isn't always about saying something, sometimes communication is to not say anything at all – yes, in some situations you might make an offer, and wait, don't bend your client's ears about how great your product is – let them take their time to decide.

I've had a potential client whom I contacted a couple of years ago that reached out to me recently, he remembered everything I showed him.

This is because people process information and make decisions on all kinds of different ways, some rely on their past experiences, some others google similar products and some others just ask around which means your reputation, people who worked with you before and how you present yourself online and in front of others is of utmost importance – you reputation and how you image yourself out there on the web and in your community is everything.

Let's talk about reputation then!

Reputation

Reputation is how you treated your clients in the past, your experiences and how you present yourself online and among your community.

You could be the fastest engineer with the best tools and highest communication skills. But if you didn't have a good reputation all of this would become a huge impediment for your business.

That's because people are more likely to decide not to buy something or enroll in a service merely based on other people's opinions than just try the product themselves.

Reputation is so, very important to the point that websites like Amazon have decided to mark product reviewers who didn't have a record of purchasing the product as a measure to make sure someone who didn't buy the product cannot ruin it's reputation for others.

Reputation could be on the product level as well as the company or the person level.

I've seen some of the biggest deals that could have been made being completely voided because of "what people said".

You want to make sure you keep all your communications professional, and on point – provide the best service you got no matter what the size of your deal is, smaller or bigger client – smaller clients bring bigger clients and bigger clients bring

you all kinds of good reputation to both bigger and smaller clients, that's why you see so many websites today bragging about their clients especially if they are one of the larger clients – put their logos on their websites even they were competitors – reputation!

Reputation could translate into marketing, but there's a trick to marketing here, which is to keep it honest, don't let marketers advertise your services or products to be something that it isn't. because lying could only get you so far before this one comment on reddit or yelp that spirals into endless amount of bad reviews that costs you your product, yourself and everything else in between!

Some entrepreneurial engineers like Elon Musk decided to go a different route with reputation – he believes if you invest all your money in building a good product, the product will basically market itself.

Tesla spends as little as $6 on marketing per each car compared to $3,000 for a brand like Jaguar for instance – they let the media and their free service which translates to reputation take care of the marketing for them – for free!

Protecting your reputation can only be done in one way, be honest, if you make a mistake admit it, own it and fix it then apologize – I see a lot of companies of all sizes try to find excuses and explain themselves, and that's such a bad way to go, because people are not stupid, they can read between the lines and some of them will dig deeper to find the truth, it's a hobby for some and a job for others like journalists, so stay honest to yourself, your craft and what you're making, combine that with good communication skills, kindness and high technical skills and you're all set!

Back where I come from, they say one hand can't clap.

No matter how smart, fast or kind you are, you need friends, partners and people you can trust to keep the ball rolling for your business.

The one man show or the savior with all the good ideas is a myth, behind every great product, company or person you'll find good partners, people one could trust with his life.

That's what we are going to talk about in the 3rd part of this article.

... To Be Continued

Behind every great product there's a team of people that share more than just common skill sets or common interest – they share "chemistry" – they figured out a way to work with each other.

They say alone you can go fast, but with a team you can go further.

No matter how rich, smart or hardworking you are, you need partners – people you can trust with your life to work with on a project of your life.

An entrepreneurial engineer realizes that you need partners in order to achieve a successful product, therefore let's talk about partners in this article.

Partners

Looking back at most of the successful startups out there, and digging deeper into the history of their success, you will find clearly that it was never a one-man-show.

You might have the most innovative idea that a human being could ever have come up with, but without partners you can only go so far before you end up in a complete halt.

Jobs and Woz, Zuckerberg, Saverin and others, Larry and Sergey and finally Gates and Paul Allen – they come in twos, threes and fours but never a one man show. Why is that?

Because having partners doesn't just complete the skill sets you're lacking, it also helps you keep the faith in your product in the darkest times, every entrepreneur out there knows that there are moments of desperation when they thought this is it, my product is never going to make it, and there's always this partner that brings them back to the arena and helps them keep the faith and continue the struggle to success.

Optimism

We need the moral support because we are not machines, we are mind, body and soul and no matter how smart the mind is and how strong the body is you still need the spirit to keep fighting and going forward to achieve success.

A smart mind and a strong body with no spirit is as good as a fancy car with no fuel.

A fancy car could have the latest tech and the strongest engines, but without a fuel it's just a useless piece of machinery, a bicycle would be more beneficial than a Lamborghini with no fuel when you need to go somewhere.

And because moral support is important, you need to make sure you surround yourself with optimistic people who believe in the impossible, the glass half full kind of folks who see opportunities instead of difficulties and always hope for the best.

In the last seventeen years in the software industry I've come across some of the smartest engineers I've ever met, but these engineers where living in corners working on things they could do with their eyes closed and their hands tied behind their backs.

When I approached these engineers to ask them about embracing entrepreneurial engineering without speaking of any specific project, I was shocked to see how internally defeated they were.

They couldn't see the bright side of anything, no hope or optimism.

I understand how useful that mindset could be when it comes to developing software.

You must think of the worst-case scenario and program defensively against it, you need to be a pessimist to build a resilient software.

But what was surprising is that they couldn't apply the same concept to entrepreneurship – the concept of seeing a difficulty and planning defensively against it, they preferred not doing it at all.

You don't need that kind of mindset when you're starting a new company or building a new product, you need to pessimists who say here's the worst case scenario and here's what we are going to do about it, not the ones who say it's impossible, it's not going to happen!

When Steve Jobs was working on project Lisa, he ran into an engineer – the best engineer his department that had a mindset of not believing in the impossible.

He never shared the vision, and although he was the smartest, Jobs decided to fire him on spot, because he was as good as a Lamborghini without a fuel!

Trustworthiness

But Partners need to be also people you could trust, blunt honest people who will give you the situation straight without any beautification!

Because when you're at the inception of a new project, you need to be able to react as fast as you can towards any changes, and in order for you to do that you need to be aware of any of these changes as soon as and fast as possible.

Your partners are your eyes and ears, they should communicate the situation to you as is, and the faster your reaction is to some issues the better the outcome might become.

But what are the rules of trustworthiness?

How do you know if someone is worthy of your trust?

I say trial and error!

You befriend your partners, travel with them, learn about their lives and start smaller projects with them, so how they would react, how they operate and how they react tough situations.

More smaller projects, little things and little situations you will get to learn about your partners, who they are and what they are made of.

The compatibility between partners and chemistry is the most important aspect of any project, that is because any software no matter how small or big it is, it is nothing but a reflection of the communications and mindsets of the people who developed it.

And if the people who develop the software don't trust each other, the outcome would be a crippled product that is destined to fail and vanish no matter how successful it looks at the beginning.

Software engineer is a very social practice, regardless of how some portrait engineers as people who live in basements and write some code in the dark – that's the furthest from what engineers really are.

Hustlers

Partners who are optimistic, friendly and trustworthy is a great thing, but if they are not hardworking nothing will ever get done.

I've met some people in my life whom I trust with my life, could throw a million dollars in their lap and walk away and know they'll bring it back.

But some of those weren't cut out for hard work.

They just couldn't do the actual work, they wanted to travel, joke and hangout but very little work they could accomplish.

You don't need that if you want to achieve something – and you can so easily tell the hustler who works long hours walks the extra mile and do the right thing when no one is looking versus those who will work just hard enough to stay in the group and feed off of the success that those who do the actual work gain.

In 2010 I started recruiting some of the students in my college to start the work on the graduation project, I intentionally set three rules to join the project:

1. We will start working on the project a year early.
2. We will meet every day at 7:00 AM.
3. We will produce code or research on daily basis.

Over 23 people wanted to join my graduation project team, but at the time the project was done only 5 were the ones who could stand still and get the project done.

A lot of people will talk the talk, believe in principles and preach them but never apply them to themselves, beware of those, because you will end-up doing all the work yourself while they enjoy their time hanging out and profiting off of your hard work.

Partnership is nearly a fair balance of effort amongst all partners, each and every partner needs to walk the whole team a step closer to success with their own way using their own skill set.

So skill set is very important for a team of partners to achieve a product!

Let's talk about diversification of skill sets!

Skill set Diversification

It's important that your partners are good at different skillsets, wear different hats and willing to educate the group about what they are good at.

Just like at war, if you were operating a tank, you need a spotter, a driver someone who puts the bullets in! – each one of that squad is doing something different to achieve the same goal.

And some skill sets are very hard to find, and if you did you might not run into the most trustworthy people who know these skills, this is when some partners decide to outsource some of the work and hire people they can trust to get some parts of the project done.

There's nothing wrong with that, desperate times call for desperate measures – you have to move forward and take your time picking up the right partners – no partners with an outsourced project is better than a project with partners that are ready to let you down at the first hardship.

Having partners is a great asset to have to build a great product, but you need to believe in your product first before any of that becomes fruitful, if you are building a product just to make profits without it being more than just a "residual income" it's very likely that your product is going to fail, there's something about the philosophy behind your product that makes it appealing a wrong that you want to right, a theory behind the action.

Let's talk about that in the next article.

... To Be Continued

The most important factor at every new product's inception is for the people developing that product to believe in it.

You have to believe that your product can do better than any alternative and will offer a unique value that no other product has ever done.

Believe in your Product

But in order to believe in a product you make, this product must have a direct impact in your personal life – something that you personally would use to solve a problem you're dealing with.

Just like school, a lot of people don't remember most of what they've studied in school because it didn't directly impact their lives, they didn't have any emotional attachment to what they were learning, it was dull, conceptual and abstract, so they forgot all about it, even if they were excellent students.

eBay

Let's look at some of the most successful products out there, like eBay, its inception and creator and how it ended up being eBay!

eBay was developed by Pierre Omidyar. Let's talk about his journey and how he ended up with one of the biggest multinational e-commerce platforms in the world.

Omidyar was a software engineer who started in the early 90s by developing software for Apple Macintosh. Then he decided to start his company with a couple of his friends and his dream project was to develop software for pen-computing.

But he and his friends weren't directly and emotional invested in pen-computing – it was just a "great idea" that they thought was going to change the world – they thought it was going to be their way to fortune.

But luckily on the side, Omidyar had another project that he was actually emotionally invested in – a project that he built for his wife who was a Pez dispenser enthusiast and collector. She was frustrated with the internet and her search to find other Pez collectors. In an effort to help his wife, Omidyar created an area on his website for her to reach out to other Pez collectors and network with other like-minded people in the collector arena.

And that was the beginning of eBay, an e-commerce website for collectors to exchange goods. He was surprised when he was notified by his internet service providers who informed him, he would need to upgrade to a business account due

to the high volume of traffic to his website. The resulting price increase (from $30/month to $250) prompted him to start charging those who used eBay, which, incidentally, was not met with any animosity.

Charging customers resulted in hiring more people, the company has grown bigger and better and today its net revenue is over 9 billion dollars.

The Secret Sauce

It's the emotional attachment and the heart in your products that constructs the "secret sauce" to making it a successful one.

Because, while we think we are so different from one another – in reality, we are very similar, we share similar interests, we feel the same feelings - fear, love, anger and everything in between, and our experiences aren't as unique as we think they are. Once we have created a product out of these personal experiences, others who have similar experiences will gravitate towards that product, because it solves a problem for them that they may or may not have been able to articulate or solve by themselves.

A product succeeds when its fully tested over and over again, and when the developer of the product is one of its users, then you have the best quality assurance engineer to guarantee that product actually works!

Developing software that you have no emotional attachment to, then testing it is a bad idea, simply because you will never be able to put yourself in the product's target user's shoes.

That's why releasing alpha and beta versions of products and targeting a portion of end-users to do some market studying became very popular as we grew and learned more and more about how to get better feedback on what we create.

When asked: "Why did you start Facebook?" Zuckerberg answered: "[On the internet] You could find music; you could find news; you could find information, but you couldn't find and connect with the people that you cared about, which as people is actually the most important thing."

Build Something You Care About

If you look at the most successful products out there, you will find a deep emotional attachment behind each of them – and that should be a fundamental part in whatever you're building.

If you pay close attention to Steve Jobs' speeches to the developers and engineers at Apple, he almost religiously repeated the same sentence: "Build something you care about".

Something you care about, something that has an emotional attachment to you, a wrong that you want to right – an evil that you want to turn good.

These phrases matter because no matter how smart the people who are building other products are, no matter how many resources they have, their products won't succeed if they don't emotionally invest and are not passionate about what they are building.

And that might be one of the reasons why startups might be considered as a threat to some of the biggest corporates – because in most corporates people go to do their "jobs", but in startups people go to chase their dreams and follow their passion.

There are so many differences between the two, it makes one person willing to invest more time and effort and push harder than an 8-5 employee.

It makes a person willing to go the extra mile even if no one was looking, they do it to apply themselves and explore how far their genius could take them. It's a personal journey, a personal product - they fell in love with the process regardless of the outcome.

They believed it's the right thing to do, even if all the odds were against them.

Elon Musk says: "When something is important enough, you do it. Even if the odds are not in your favor"

Conclusion

This concludes my series on entrepreneurial innovative engineering - things I've learned and wanted to share with the world. And I have no shred of doubt there's still a ton more to learn out there. It's all about the journey of an engineer from the moment they write a "Hello, World" program to the moment they release their million-user product that changes the world.

This is my documentation to the patterns I've seen being in the industry myself, from design to development and from architecture to deployment ... and beyond.

CHAPTER 25

Systematic Innovation

Innovation at its purest form is to break out of the regular and settle for no other than the extraordinary.

Innovation is the reason why we can talk to virtually anyone in the world today and get a response back in milliseconds.

Innovation Kills Itself

But innovation could also kill itself. It could cause the process of its own creation to slow down, it's a very strange relationship because the larger the innovation the lesser people would start thinking of better ways to accomplishing the same task, differently.

I have always struggled with the question of how to maintain the same level of innovation that has driven geniuses like Thomas Edison with the light bulbs, or Karl Benz with the cars and many, many others who changed the world as it used to be, to what it is today.

I have come to an understanding that innovation needs to be fostered, nourished and invested in, in order for it to maintain it's levels, the very same levels that driven earlier innovators to make our world what it is today as we know it.

Part of the Process

But for innovation to continue and grow, it needs to become a fundamental part of the fabric of every development process, regardless of the industry.

I will speak for software development industry, but it surly applies to many other industries.

It's very often for software engineers when trying to solve a problem is to jump to the most familiar solution they have done in the past for similar problems.

This very action, settling for what's familiar is an anti-innovation approach to solve problems, simply because settling for what's familiar doesn't move the needle any further when it comes to solve that very same problem.

Think about it this way, if people in the 1800 settled for settled for paper publications, we wouldn't have digital media today, if people in the 1990s settled for landline phones we wouldn't have cellphones today, same thing goes with PCs, electric cars, airplanes and so many other inventions that we use today almost on daily basis.

Innovation should be a mandatory task on very agile board to solve every problem, to think differently about solving a certain problem, regardless of its size or complexity.

Mini Hackathons

It's like having a mini hackathon where developers take an hour or so with every task to think about different ways, they could solve a problem they already know it's traditional solution.

These mini hackathons help engineers always look for different ways to accomplish their development tasks, it's a daily training for their minds to think out of the box and try to tap into the *different*, no matter what that *different* looks like, attempting to access that portion of our brains where we trigger imagination, inspiration and unveil the unknown.

It's a Skill

Innovation, just like any other skills requires training.

You don't become a software engineer in a day, you go to school, you read books, you watch tutorials and clone projects and experiment with them, it takes time, training, making mistakes and learning from them, then making even more mistakes.

It's the very same thing with innovation, it begins with asking yourself, do you just want to do what everyone else is doing, or do you want to do more? Be more?

Everyone has it

Every person in this world has at least one innovative idea that could change the world as we know it. Some people keep knocking on that door until they find their genuine idea, some others don't even try, they don't try because somehow, somewhere someone convinced them that they are not capable of innovating, that they don't "have it".

They are all wrong.

They are wrong, because every innovator out there has been told something very similar, Walt Disney's manager told him once that he wasn't talented, Walt Disney has become one of the most creative individuals of this century, and for centuries to come.

Why Innovate?

But someone might ask, why innovate? Why bother? Is it the wealth? Is it the fame?

I have been studying the history of innovators very closely for years, and what I have realized is that a lot of them didn't really seek fame or wealth, it was the very process of innovation that made them feel they could apply themselves into something that is timeless, something helpful, useful that could change people's lives for the better.

Because every one of us, every single individual out there has that one unique idea that could help us all be better, the very process of innovation is a process we all should be participating in, it's a process that we all need, and we all have the capability of accomplishing.

Are you Automatable?

For software engineers, every process that can be automated is a process they should be applying the innovative aspect of themselves in. in other words, every engineer out there should be asking themselves when solving any given problem, what would the robot be *not* able to do if this problem was presented to some autonomous problem-solving platform?

It's a Social Activity

Innovation is also a social activity, bounce your ideas off of someone very opinionated, and very straightforward with their feedback, try with multiple people, gather some people for coffee or lunch at work and throw an idea in the middle and see how far a *distributed system* of "bio-processors" could do with your ideas.

Don't worry to much about folks who shutdown your ideas, think it's not useful or meaningless, listen more carefully to those who open your ideas to different aspects, different venues, these are the ones that can help you go further faster with your ideas.

I have seen some engineers stay away from opinionated engineers, they should do the opposite, to get perspective, feedback, regardless whether their opinion can affect the decision making of some process or not, that doesn't matter, and it shouldn't be the goal of the discussion, and in these discussions listen more, listen carefully, sometimes people get too excited they interrupt each other, make sure you keep track of each idea then follow up with people individually to learn more about their ideas and it entails.

The difference between us as human beings and some AI-powered system is that we need to apply ourselves, the aspect of inspiration and imagination that is almost impossible to automate, this is where innovation comes from, something that seems *illogical* at a time, but then becomes the norm once accomplished.

Embrace the Different

Embracing the weird, the strange, the insane and the unfamiliar is a good start, but it needs to become a habit, a daily habit with every single task, let it be design or development, testing or deployment, regardless what the task is, do it differently, think differently, don't fall in line, it's only the dead fish that goes with the flow, don't go with the flow, follow your imagination, see how far it could take you.

CHAPTER 26

Code Rub

In every software project I've ever been involved in, there was always this issue with code maintenance.

Newer practices, language versions or frameworks are released, and a year-old project become a legacy so fast due to code rot.

In the past, I heard some people say, we don't have to have the latest and greatest, we don't have to learn the most recent best practices or code styles, some people say: we're fine. If it works don't fix it!

These are the kind of engineers that decided to stay behind and settle for less, and I promise you, given enough time these very same engineers will have no clue how things work anymore, all they'll have is some tales from the good olden days or fight you ferociously when you try to introduce newer frameworks, language version upgrade or anything related to modernizing the software.

When this kind of has-been engineers are in power positions, due to their seniority not their leadership skills, they become an obstacle in the way of innovation, modernization and the contentious growth of the poor souls they manage.

Because modernization inspires continuous learning, innovation and keeping up with the times, and all of that requires effort, and in the software industry if you're not wholeheartedly into engineering you'll run out of breath so fast.

But staying up with the times requires it to become a fundamental part of every engineer's daily work.

I have come to realize that a systematic approach is needed to make sure the team is in a continuous state of learning and innovating while still being able to deliver value.

This very practice is what makes an engineer feel valued, invested in, and makes them willing to stay longer at their positions because they are getting more than just a salary or a benefit out of their job.

That's because investing in the continuous learning of engineers is just as important as their salaries, simply because their continuous growth is the only assurance they have to ensure they will stay relevant in the industry and continue to do what they love to do without wasting eight hours of their day working on some outdated irrelevant technology or skill set that has no use for them anywhere else.

And because of all that, maintaining software has become more important than just keeping the software running seamlessly and decreasing its faultiness, maintaining software becomes more of a maintenance process to the knowledge of the engineers working on this project as well.

Here's some of the requirements I needed to achieve when I first thought about code rubbing:

1. Engineers need to stay relevant, up to date with technology and coding best practices to produce better quality software.
2. Engineers need to develop a sense of pattern for the consistency of code, and potentially to develop their own pattern and best practices.
3. Engineers need to familiarize themselves with every single file in the solution to be able to pinpoint any potential issues in production and be able to rapidly fix them.
4. The solution itself needs to be in a continuous state of maintenance to ensure its relevancy in the market and betterment of user and developer experiences.

The order of these requirements is intentional, because good engineers produce good software not the other way around, it doesn't matter how advanced modernized or up to date your software or your process is, if you hire bad engineers they'll turn it into a Dutch oven as fast as the rates of gravity.

So, engineers here are the top priority, and therefore I devised a software maintenance strategy that I called code rub.

Code rubbing is just like glass rubbing, it makes cleaner, shinier and better to look at.

In order for that to happen, every engineer will need to pick up a random file from the solution they are working on, and make sure it follows the best practices and latest technologies, along with making sure it maintains it's consistency across the entire solution.

But there are conditions for code rubbing, which are:

1. It shouldn't take more than 20 minutes from an engineer's day.
2. It should target only one single file per day.
3. It shouldn't include implementing new features or fixing bugs.
4. It should require more code reading than code writing.
5. It shouldn't require a pairing session.
6. It should be done continuously every single day.
7. It shouldn't happen to the same file by the same engineer twice unless every other file has been already rubbed.

Sometimes the file you pick up might not require much change, in that case an engineer's code rubbing task is done for the day and they are ready to work on their daily tasks.

Code rubbing inspires conversation among the engineers to find better ways and learn from each other, conversation result in innovation when egos and pride are set aside.

In a team of seven engineers code rubbing a solution means that every single file in the solution has been reviewed, maintained and read at least seven times before it was beta released.

And the more an engineer rubs the same code the more they learn about it, it's like reading a book more than one time, the more you read the same book, our brains automatically skips through the obvious and looks for the unfamiliar to learn from, so is the case with the code.

But code rubbing also means that every engineer will be producing a new pull request every day, pull requests make engineers happy, especially when they are working on a long running sprint-long task and they haven't merged anything in a while, code rubbing gives engineers this feeling of accomplishment and belonging to the project, especially newly joined engineers.

In an Agile methodology, every engineer should have a code rub task for every day in the sprint, a two weeks sprint will require ten tasks of code rubbing, whether the code requires a rub or not, creates a pull request or not, the task moves to completed just for the mere effort of reading through the code and making sure it's in shape, regardless how big or small the file is.

There's a mindset in the industry that code must be written right from the very beginning, it has to be perfect from day one. But the reality is that code is perfect only for a short period of time until a better practice, technology or a language version is released, that's when the code immediately becomes outdated, faulty and harder to maintain for newer engineers.

In contrary of what Robert Martin (Uncle Bob) says, that code rots because people who are working on it are rotting it, but in reality code actually rots by itself due to the continuous state of advancement in technology, not just by bad engineers not following practices and regardless of what the cause of rotting is, the goal here stays the same, code rubbing to ensure cleanliness and relevance.

Multiple teams and myself have started the code rubbing practice in the past several weeks, and I have to say the results have been inspiring enough to share this experience with the world in this article, inspiring enough to recommend it to every software development team out there given the team have already settled on what their coding standard will be, and how often after a release they are willing to upgrade their libraries.

A software development team without a coding standard is like trying to fly a spaceship without a manual, it's not a good idea, and if your team doesn't have a standard, they won't be able to perform code rubbing, simply because what one engineer sees as a best practice, another engineer will come later and undo their work, it will be like trying to fill a broken jar with water, it retains none, and it's a waste of time.

At the end, I like to think of code rubbing as a product of innovation, systems that we devise due to our accumulative experiences for the betterment of the software development process, and as we continue that practice, I'm sure we will learn more along the way and continue to produce better systems and practices, adjust our existing processes as we go to help us get to where we are going faster.

CHAPTER 27

The Tribe

In the software industry, the term tribal doesn't really reflect a good organizational structure, for instance, the term tribal knowledge, means that a given software is poorly documented and its details are basically fully based on what one engineer said to another.

I have personally long struggled with the term and considered it to be a sign of an alarming pattern when it grows within a team or an organization.

But the longer I learn about our industry, the more I realize that tribalism is a natural human behavior, we tribalize as humans because we want to belong somewhere, let it be a group of individuals, or around a certain idea, belief or practice.

Regardless what the cause of tribalism is, it seems that we as humans will always find a cause or a reason to aggregate and form tribes, you can see it clearly at all stages from children gathering together to play a game, to adults forming clubs to a larger scale such as the common wealth, the Asian Dragons (Four Asian Tigers), the European Union and so on.

But tribalism just like any other form of organization can be either a force for good or a reason for divide. For instance, belonging to a group of individuals might instill the feeling of superiority above all "others" or it can strengthen a good cause and willingness to help others and aim for their happiness and prosperity.

It's like the difference between an organization set to spread lies and harm others to gain power, and an organization that gathers donations to help those in need and promotes the values of selflessness and sincere willingness to improve the lives of others.

So, if that is the case, it might be then safe to assume that a tribe's value or the natural practice of tribalism really depends on what entails not necessarily the act itself.

I thought long and hard about what would make a good tribe, and I came up with the following:

Welcoming

A good tribe is welcoming to all people, accepting of all people, for as long as they don't reverse the notion of welcoming, joining a welcoming tribe implies that one is willing to also be as welcoming (or trying to be) to others.

A tribe that welcomes unwelcoming individuals is a self-destructing tribe, it would be a matter of time before those who are not welcoming to others become a majority and the entire purpose of the tribe takes the opposite turn, and becomes a force of destruction and evil rather than goodness and prosperity for others.

A welcoming tribe does not just welcome others to belong, but also welcomes their perspectives, ideas and approaches and works to learn from the newcomers as much as it is willing to teach them.

A welcoming tribe does not really care about the seniority of a person in a tribe, their influence or position. it does not consider those superficial measures to consider an opinion of an individual, it takes every opinion, approach, and methodology just as seriously as any other.

That is simply because a welcoming tribe understands that the "right way" or the "truth" are not exclusive to a particular individual, an innovative idea could originate from a simple person sitting in their garage in a suburb, innovation definitely doesn't care about how long you have been in the industry, or what your title is, neither what education degree you have earned.

Selflessness

A good tribe puts others before itself, this can manifest in so many different ways, my personal favorite is when two engineers are pairing to develop a feature, and another team member requires help, to review their code, or help them understand the business or even seeking a simple advice.

When engineers stop what they are doing, and give their full attention to others, this behavior becomes the most optimum for an engineering development process to run in parallel, because when you unblock another member you allow them to continue to work in parallel with you, they won't just be sitting their idle waiting for you to help them while you go on your way and finish your feature work.

Selflessness is the most optimum path to achieve a goal when one is a member of a team, group or a tribe, because selflessness guarantees an ongoing reciprocating gratitude that keeps getting larger and larger within the group, until it becomes the norm to selflessly give away some of one's time to help others and it becomes the norm to be grateful for others to help you out.

Selfless is also infectious, a selfless act inspires another, it inspires one to do the same and pass it forward, these values dictate on every individual in a good tribe to care about people maybe they will never meet, especially the ones who can't pay the favor back, that's the ultimate act of selflessness.

Sharing

In so many occasions, every time I tried to share something new I learned about a new framework, technique or pattern in developing software, I always received an even more enlightening perspective from the people I share the new patterns with.

Proactively sharing knowledge, is another important aspect of a good tribe, reaching out to others and talking to them about something you've learned, it conveys the message that you care about them genuinely that you want them to know as much as you do, and it also inspires them to do the same with you and others within the tribe.

Sharing is caring they say, truer words have never been said.

Because when you share what you know with others, regardless of whether they have anything to share back with you or not, you express the ultimate value of sharing.

Sharing engulfed with humbleness is the way to get your point across to others, because sometimes sharing with others could be intimidating for them, it can be misinterpreted as flaunting one their knowledge on others, sharing without

humbleness can be interpreted as no sharing at all.

Flexibility

Every tribe, let it be an organization, team, a startup or an entire corporate have some form or another of rules, protocols, and regulations.

These rules, protocols or regulations help streamline the process of the work the tribe is set to do. In the software industry, these rules become more and more important to ensure the consistency of code, and the healthiness of the services being developed, and the cohesion of the interfaces being designed.

But a good tribe doesn't hold on to it's rules and standards as if it was a divine text – every rule, standard or protocol created by man is subject to change with respect to the time, place and the million other circumstances and variables that makes a rule subject to change.

"You have to understand the rule, so you know when to break it", my friend David always said.

Rules are meant to be broken; a good tribe understands that its rules are always subject to change if better ideas are proposed.

A tribe that idolizes its founders and perceives their rules as divine is not a good tribe, because idolization usually leads to conflict when a better way is proposed, because a bad tribe thinks that an opposite direction even when it achieves the same epic goals of the tribe is proposed they see that as insulting to them and what they believe in.

A truly good tribe is willing to explore the unknown, in fact, seeks it and learns from it.

But with flexibility comes the firmness of not allowing anything or anyone to deter a tribe from it's major goals, in the software world, the dedication to the quality of the product, and betterment of the experience are non-negotiable aspects that shall never be changed, a good tribe holds on to that, regardless of the methodology, rule or protocol that leads to achieving that goal.

Empathy

Loving others and caring for them is a skill set, just like any other.

Loving others manifests itself in so many ways and shapes, it can be generosity, kindness and sometimes empathy for others.

A good tribe empathizes with others, understands their circumstances, and seeks to excuse them when they fall behind – because life is not just what we do, but also what happens to us. And what happens to us can slow us down or prevent us from doing what we want to do.

There is a big difference between those who do not want to fulfill their purpose because of the circumstances and those who don't because they don't see it as a purpose in the first place. The first type is those willing, dedicated and committed to the cause but something out of their control has blocked them, and for those we shall empathize, forgive and be understanding of their situations.

But those who have lost their way, we owe them the responsibility of helping them find their way back, find their purpose and their cause. The hardest of decisions is to decide that there is no coming back, and it's time to move on and hope for those who lost their way to come back and be a part of the tribe again.

Tribes come in all different shapes and sizes, we all them teams, startups, communities or corporations – we call them family, club or any other names but at the end of the day these are nothing but tribes, they have their own written and unwritten rules and protocols, and the goal isn't to shame that natural act of belonging but to be able to understand and identify the destructive aspects of a tribe and potentially find the way to turn it into a force of good into people's lives.

Corporations today have failed to understand that a good act of tribalism can serve the overall goals and ambitions of it's leaders faster than anything else, they choose to shuffle their employees around and reorganize an entire structure to keep the ties loose to prevent any potential unionization and a well-formed opinions about certain protocols and practices the corporation may follow.

This is when the corporation itself chooses to be on the wrong side of tribalism, the side that prevents a sincere well-informed hive-minded force to lead the way to a better future and a sooner success regardless what that success is or how it's

defined.

But a good tribe could be as little as one man, a welcoming, sharing, open-minded, flexible and empathetic man *is* the good tribe, and a good tribe is a magnate to others to belong to the one man, growing in size and power paving the way for others to follow suit and work for the greater good for everyone regardless if whether they are a part of the same tribe or not.

For those who read my words, I say, if you don't see a tribe then become one, become a good tribe, become the reason people aggregate around you and follow you to help them learn and understand how to be better, happier and willing to spread that happiness and prosperity to others.

CHAPTER 28

Sandcastles

Sandcastles are models of castles, built out of sand usually by children.

This is how Oxford dictionary defines them.

In the software industry, some software might be just like sandcastles, very beautiful on the outside, but extremely fragile on the inside, built by immature engineers.

The biggest intellectual plague that might infect an engineer's mind in the software industry is when their goals and the quality of the software they build do not align together.

Building a project just for the purpose of getting a reward, attract an investor or earning a promotion to move on to the next role leaving behind sandcastles of software that requires a complete rewrite.

I have always promoted the idea that rewards, investors, promotions, and customer appreciation are side-effects of building high quality software, not the other way around.

Because when the reward becomes the goal, an engineer might develop the mindset to work just hard enough to earn that reward, regardless what happens after – it diminishes the integrity of the software being built and the effort being spent to engineer an industrial high quality product.

In the past, I have seen these sandcastles becoming the burden of the unlucky of the newcomers to the project to maintain a fragile structure, spending days and nights trying to make sense out of its unnecessary complexity – questioning their very skills and abilities to provide value, fearing being accused of inexperience and immaturity for not being able to hold a falling structure.

Any software that does not follow a consistent pattern, is doomed to be rewritten from scratch, regardless of how long it takes those who don't have to maintain it to realize it doesn't serve their high goals anymore, regardless of the suffering of those beneath them who sacrifice their time and efforts to maintain the sandcastle from falling.

Sandcastle software has obvious attributes that should be visible with the naked eye regardless of the years of experience an engineer might have, so here they are:

Clone, run ... FAIL!

Some of the most obvious things I have seen through the years, is that sandcastle software never runs as soon as you pull it down and try to run it.

There is always this little configuration here, or permission there that takes from days to weeks to just get you to hit the *play* button and start dealing with the next nightmare.

An industrial software should have the proper documentation and guidance to get a newcomer started and to save some of their time from being wasted asking around trying to figure out how to get to run the software.

Documentation is important, but more importantly the team's willingness and proactiveness in getting a newcomer started and comfortable by giving them a tour of the software so they know how to push features and resolve issues whenever they occur.

Testing

A sandcastle software also lacks testing, and if it does have some tests, they are a mesh mash of multiple concepts and ideas, with lesser code coverage and several blind spots that gives a false notion of quality, but then again just like a sandcastles, it couldn't be more fragile on the inside.

When it comes to sandcastle software, you will notice the following issues with the testing:

1. It might not exist at all.
2. Half-baked tests with false positives.
3. A lot of commented or disabled tests because of whatever reason an engineer could come up with that day.
4. Blind spots, where a block of code could be completely changed, and no tests would raise any warnings or issues.
5. Test complexity, unorganized hard to understand tests that require their own on-boarding session just to realize they are nothing but bloated overly optimized tests for no good reason.
6. Running the test requires an entirely different set of skills and technologies just to get a proper status.

Mishmash of Patterns

In a sandcastle software, there is no cohesion between the components of the system.

It becomes a free for all zone where everyone coming in, using their own patterns to implement a given feature – in that case a software becomes a lot like a book written in multiple languages, talking about different topics with no cohesion or story-line.

You can see this very visibly when you see multiple methods doing the same thing (mainly helper methods) written by multiple engineers.

When the software being written and maintained lacks cohesion, readability immediately suffers immensely which inevitably leads the engineers working on the project to lose the willingness to read the code, and focus more on just getting a feature implemented regardless of whether some of the code being written already exists and can be leveraged or not.

These are three very prominent aspects that I have observed over the years in sandcastle software, but they are not the only ones.

Some software is simply a reflection of the culture and communication structure the team follows – a team of isolated engineers not pairing and not communicating should eventually produce an out of sync code that has multiple redundancies and inconsistencies across the board.

I have pointed out in multiple occasions in my articles and video sessions that software development is a social activity. It requires every engineer to communicate with the rest of their teams to solve problems and agree on the road-map and the road ahead.

When these communications become exclusive to stand-ups and planning meetings the quality of the software suffers, and the success of the project becomes more and more harder to achieve.

The *Scope Triangle* dictates that when it comes to cost, quality, and time you can only choose two of the three, and be willing to pay the cost of the third.

For instance, if you choose giving the engineers all the time, they need to build the highest quality they can possibly produce, the time aspect of the project will be longer.

The same thing goes with time and cost, a project with the lowest costs and fastest delivery time will result in the quality of the produced product to be very poor and almost unusable.

But these three aspects are from the perspective of the stakeholders who think quality is a negotiable term. But from an engineering standpoint, quality is actually a none-negotiable aspect of any software being built, it insults the very integrity of true industrial engineers to negotiate the level of quality they want to build their software with, it's not an option to talk about.

Which inevitably leaves the business non-engineering crew with one or two aspects, either they choose the time and bare the costs, or choose the low costs and wait longer to get the product delivered.

But when engineers agree to negotiate on quality, this is when sandcastle software is born, and while the business is happy to have "met their quarterly goals" – the reality is that what has been delivered is worse than delivering nothing at all.

Years ago I could see the look of disappointment, confusion and shame in the eyes of the engineers delivering software they know full well that it doesn't live up to the level of celebrations and rewards being spread around over the product they know is doomed to fail.

True integrity dictates that one shall do the right thing even if no one is looking.

Even if you have the rights to push straight to the main branch, run your deployments and call it done.

True industrial engineers have their own internal code reviewers, the voice of integrity in their hearts that calls upon them to do their very best regardless of the pressures the stakeholders may enforce thinking it will make things go faster.

Those industrial engineers are the only ones to be entrusted with the future of this industry, simply because the more we head towards the future the more critical software becomes in our daily lives.

Software today is being used in some of the most critical situations to make the best decisions to save lives or prevent dangers. and if the engineers entrusted with building that software only cared about pleasing their business owners, delivering the fastest and seeking the rewards regardless of the quality of software then the future of humanity is doomed, and a serious revision is mandatory to the principles that run the industry.

These industrial engineers are our hope for a better tomorrow, their selflessness is observed the most when they *willingly choose* to spend the extra hour and go the extra mile to ensure everything is in good shape.

Without these engineers the entire industry would collapse, trailing tribes of sycophants who care more about their earnings, rewards or promotions than the true value of what they do and how they do it.

Some engineers build industrial software to seek quality, and they so happen to get rewarded for it as a side-effect.

And some others build sandcastles only to seek these very rewards.

CHAPTER 29

Unbiased Compilers

A line of code written by a junior engineer is just as effective as a line of code written by the most senior engineer out there.

To the compiler, it either runs or not. Regardless of who typed it in, who designed it and who hit the "run" button.

One of the biggest challenges an observer might see in the software industry is that the industry did not quite fit in the conventional way of industrializing a profession.

It's very common amongst software engineers that delivering a feature or fixing an issue doesn't quite fit the regular 8 – 5 work hours or even the environment where an engineer works – sometimes stepping out of your desk for a walk for an hour is a part of solving the problem you are working on.

That is because a software engineer is an artist, thinker, philosopher, and a negotiator all at the same time.

A software engineer is required to visualize solutions almost all the time, user interfaces, patterns, high- and low-level architectures and even the symmetry of the code written all at the same time.

A software engineer is also a thinker for a living. Solving problems day in and day out is a fundamental part of this craft, and it comes in all different shapes and sizes – managing memory resources, optimizations, or simply making something logical and easier to maintain and understand – problems are everywhere, and so are their solutions.

Being able to effectively negotiate and communicate ideas to both very highly technical people like the engineers on your team, then switching gears to explain the problem to a usually non-technical folk on the business and management side

to get your point across. It requires a tremendous effort of being able to speak about the same thing in two sometimes three different languages to get everyone on the same page.

But luckily, these skills are not really bound by a person's title or how long they have been in this industry – I have seen some of the most beautiful software written by folks who would be ranked as "juniors" or "entry levels" in the ways the industry ranks an engineer today.

The biggest proof is that the most popular software we use today was all started by engineers who had no experience at all, they built it in their garages and their dorm rooms like Google, Facebook, Microsoft and so many others – it was mostly built by engineers in their early 20s the ones we would call today "interns".

And the skills that I mentioned earlier are also not bounded by how long someone has been in the industry either, in fact the newer a person is to the industry the more likely they are to bring a new perspective that would redefine the standards and pave the way for a whole new way of how we look at our problems and how we solve them.

Compilers continue to remind us of this universal truth about our industry – compilers don't ask the person hitting the "run" button how long they have been in the industry or what rank they are in the place you they work, compilers look only at the code written and decides whether it works or not, fast or not feasible or not – and that's all, and that's it.

I believe that any measurement of someone's pay grade, rank or level of appreciation based on their seniority, experiences or whatever they have written in their resume shall be a completely fallacious if not fully influenced by what they actually can and cannot do – which can only be decided by one thing – the compiler!

Some software companies have come to that realization, so they started properly interviewing software engineers by doing katas or fully days of pairing with the person, to get a full grasp of how they solve problems, deliver features, how fast they adapt and how fast can they think on their feet and deliver value.

Every other measure shall be just a nice-to-know or a nice-to-have but nothing that should have any impact whatsoever on the final decision of whether someone shall be hired or whether they qualify for a particular title.

When we hear the term Ageism we think about older individuals being treated unfairly due to their old age. But ageism could also manifest and show it's ugly face when a junior 20-something year old engineer is being paid less despite the fact that they answered all the questions, solved all the puzzles and did a rain dance on top and yet still get paid less and ranked lower just due to his "experience".

It's another way of saying, you are very smart and you know all the answers, but you're young so you shouldn't be paid your intelligence worth, you should be paid your age worth – that's a hidden form of ageism.

Our software industry requires a serious restructuring of how we measure the skill-sets of engineers and how we evaluate an engineer's work and effort and the true link between what an engineer brings to the table and how they are appreciated and rewarded for it.

An engineer reward or rank should not in any way be influenced by anything other than what they create – therefore what is being created shall have proper measures for evaluation.

This makes the equation so much easier. Because measuring the creative work of an engineer is much easier than trying to measure the engineer himself.

Simple things like, does the code run fast enough? is it clean and easy to read? Is it modulus and pluggable? Does it cover all edge cases?

These measure is what should dictate a person's rank, rewards or position at any company, these are the measures that will help the industry assign the right talents in the right places, and help us all move forward towards the next phase where we become more focused, more precise and more fair about how we treat each other and how we invest in the growth of each other by learning from the talented amongst us.

As a person who conducts interviews on monthly basis, I highly urge everyone who conducts interviews to focus on the skills and the technologies not how long someone has been in the industry, these measures mean absolutely nothing if not

accompanied with actual talent and solid skills not just tales from the past.

CHAPTER 30

Pure Coding

In any software development process, there are multiple activities that are involved in the process of creating software.

For instance, there are design sessions, architecture, development, testing and deployment amongst many other activities that are crucial to building a successful software.

In this article, I'm going to talk about the process of developing software, specifically speaking writing code and contributing this code to a source control medium to ensure the code is stored at a centralized server where everyone else in the project can copy it, use it and build on top of it.

git

In the software industry today the most dominant source control medium being used is git – which is a distributed version control system for tracking change in the source code of any project. It was designed by Linus Torvalds to coordinate the work amongst software engineers to ensure no loss in code may occur and ensure everyone is working with the same code base.

But when developing software, it's important to engineer not just the software being built, but the very process of developing the software itself.

Common Practices

The way I've seen some features being developed and committed to source control today is that it goes from WIP to DONE without any indication of how the software was actually written and the code was developed.

The problem with this practice, especially in a test-driven environment, is that there is no way the history of the code being committed could indicate whether the tests for this code were written before or after the implementation code was actually written.

This becomes even more problematic when engineers rely on tools like code coverage to use as definite indicator of whether the code is fully tested or not.

I have also seen more and more declining interest in being descriptive about the code commits, and what they actually indicate – which makes it even harder to find a particular feature implementation commit, when it was committed and who the subject matter expert to discuss code changes with.

Coding is a Social Activity

Coding just like any other activity that involves more than one individual, is a social activity. An activity where the success of it relies heavily on the proper communication between the individuals involved in that activity.

In software development, committing code with proper messaging and content is a key to the successful communication amongst software engineers developing the software.

When the communication is clunky, unclear or vague, the entire process faces the risk of being lossy. Lossy in the sense that some information will be lost, some features will be hard to traces, and some engineers won't get the credit and appreciation they deserve.

When I look at any code repository, I can very easily determine the level of communication and cooperation between the team involved in developing in that repository so easily by simply looking at the commit history.

Working Code is Not Enough

Some engineer once told me: "It doesn't matter how the code was committed, what matters is that it works!"

And I couldn't disagree more with this statement, simply because a "working code" isn't just the code that fulfills a particular business need, but also the code that is easy to maintain and easy to understand.

The maintenance and understanding of any code base rely partially but crucially on the history of how this software was developed, and who was involved in the development process of that software.

Especially in enterprises where engineers might get shuffled around often or move on to other teams and some of the domain knowledge that governs why particular features where the developed in a particular why.

A given piece of code could tell you how a particular goal can be achieved, but it can't tell how why that goal needs to be aimed for in the first place. This is why communication is crucial, documentation is crucial and being clear and explicit about the purpose of your code and what stage it was in when it was committed is even more important than most other communication activities.

Pure Code

And because of the importance of code history, multiple teams and I have decided to purify our code history, by making short commits to describe the development process of each and every feature we are developing.

In other words, when an engineer writes a failing test, they have to make a commit, that puts out a clear title of what that failing test is, and that its status is that it's failing as follows:

ShouldRetrieveStudentByIdAsync → FAIL

The next commit should indicate that the very same test has passed, and there should be no other code involved in that commit except for what makes the test pass, as follows:

ShouldRetrieveStudentByIdAsync → PASS

A commit history that follows that pattern is what I call Pure Code, and the activity that produced it is what I call Pure Coding.

It's pure and beautiful simply because it's clear about its origins, how it was developed, who developed it, and what their thought process while they were building it.

It makes it even easier for other engineers on the team waiting for a feature to be developed to have a real-time track of where the code being developed is at, looking at a DRAFT pull request.

It opens the door for another form of well-engineered communication methodology of letting other engineers on the team know which direction a particular feature is being developed in and where it's headed, therefore allows even faster feedback for other engineers to reach out and help with some insights before a particular feature development goes off-rails.

Pure Pair

This pure coding approach, becomes even more meaningful and magnificent when two engineers are pairing on the same feature – it follows the following pattern:

ShouldRetrieveStudentByIdAsync → FAIL [by Alice]

ShouldRetrieveStudentByIdAsync → PASS [by Ken]

ShouldThrowExceptionOnRetirveIfStudentIdIsInvalidAsync → FAIL [by Ken]

ShouldThrowExceptionOnRetirveIfStudentIdIsInvalidAsync → PASS [by Alice]

You can easily see in the above history, the *ping pong* pairing pattern at its finest as the history of the code itself is telling the story of how two engineers built a particular feature in a TDD approach and ping-ponged the development between them until it was finished.

Communication All the Way

Pure coding clears out all the miscommunications that may occur in a development team about understanding the pain points and the complexity of developing a particular feature.

The most common issue that happens is when a feature is delayed, or surpasses it's expected delivery date.

Naturally questions will be asked why a given feature is taking too long?

Pure coding makes answering this question much easier, it even makes tech leads have better understanding of where pain points are when developing a similar feature and helps them work with data to streamline the future development process in similar future features.

It also makes the estimation process of features even easier and more accurate than others.

And so is the case with the standardization and adaptation and commitment to any pattern, it helps streamline the processes and when cultivated it becomes a guarantee for a resilient industrial software development teams, and more communicative software engineers.

The Future

These patterns and practices that we develop, help us shape the future of the industry, as our software industry grows, more measures and standards have to take place to ensures the quality of the software we develop. Especially that it's becoming more and more involved in our daily lives, and more involved some of the most crucial decisions being made about our future.

It's not enough to write code that works, and it's not enough to write a beautiful piece of code, it's also important that very process of committing the history of that code to also be just as beautiful – that's software craftsmanship and that's what every engineer should aspire to achieve in their journey in this industry.

CHAPTER 31

Reusable Modularity

In the software industry, there are two popular schools of thought amongst many others when it comes to designing software.

A group of engineers will lean towards stateless modularity. They will build each component of their system as a pluggable and configurable agnostic module. And to ensure the independent nature of the module they might run into some seemingly redundant pieces of code to prevent any hard dependencies on one centralized component.

This approach is useful in the sense that it guarantees avoiding all possible single points of failures from a dependency standpoint.

It also guarantees that all team members involved will have the opportunity to build similar components and learn from the process of building an entire pipeline for one feature or another which shall eventually have a great positive impact on their growth and success in this industry.

It also comes with both good and bad impact from a cost standpoint.

On one hand, writing seemingly redundant functionality for separate components will result in an easier and gradual adaption for newer technologies and best practices without having to bear the cost of upgrading everything at once to match the new technology enforced by the reusable component shared across the system.

But, on the other hand it also means that there's an additional cost paid at development time when each and every engineer have to write a seemingly similar functionality such as validations for instance which shall eventually add up to the overall cost of the software being developed.

Another group of engineers will be more leaning towards reusability. They will generalize and package every single component they see reusable to cut the costs of having to develop and write similar functionality in several components throughout the system.

And that approach's good side as obvious as it may seem is very effective from an economic standpoint during development-time, as more components are being created rapidly without having to bear the cost of what seems to be redundant functionality such as validations for instance.

The problem with that approach, however, is that it creates bottlenecks and single of points of failure across the entire system.

It also enforces a full payment of tech-debt at once as soon as a new technology is adopted or a new standard has been enforced which sometimes results in engineers losing interest in maintaining the software and declaring it a "legacy app".

So, someone might ask, what's the best option here? Shall we modularize across the board and ensure gradual growth for multiple components or save the initial cost of development by publishing reusable packages while planning for potentially high tech-debts or full rewrites of entire systems?

The answer as it usually is it depends.

If the modularity is enforced on some external non-business routines then it might make more sense to package that up and turn it into reusable component.

For instance, communicating with a certain external resource, if we continued to redundantly write the same authN/authZ routines to communicate with the same resource then that'll just be a plain waste of time and effort – I would side with the reusability team on this instance as it doesn't really make much sense copying the same brokers across multiple components.

A more material example would be the remapping of native exceptions originated from interfacing with a database would be much more economical if it was packaged and distributed for multiple target frameworks without trying to generalize some of the business-specific operations brokers may use to leverage said database functionality.

But on the business logic side of the system, an enforcement of a reusable package would be disastrous as it threatens the very core of what the system is developed to be.

Especially with the current state of the world when packages have to be referenced and pulled from their external repositories on build and deployment, a sunset of a particular artifactory would bring the entire system to its knees and cost much more to remedy.

But the same situation with brokers would at it's worst case scenario simply require a re-implementation of the basic non-flow-controlling operations that brokers may have without touching the dense part of the system, meaning the business logic.

The same concept applies to public endpoints and controllers in a RESTFul system – but it shall at all cost be avoided for mid-layers where the decisions are being made about any particular incoming request or outgoing response.

Enforcing modularity on the mid-layer portion of any system allows more team members to experience in practice how particular services come to life without having to go through poorly documented tutorials about abstractions that have no existence in any world outside of the current project.

When reusability of logic is enforced in a larger system, sacrifices have to be made to maintain the generic nature of a packaged routine – which shall eventually change and require a deprecation or inconsistency across the board.

You can very easily spot this pattern when seeing a bloated shared package that has many routines that only serve one particular case in a single business logic layer *hoping* that it may be reused and needed in another flow at some point – it never is.

In conclusion a hybrid approach of business components modularity alongside reusable primitive operations seemed to gather the best of the two worlds to ensure the ongoing growth of the development team, being more adaptive to a forever growing and changing industry and following a more economically efficient development procedures than any of the two approaches individually.

CHAPTER 32

Time-Travel-ish

If you take a closer look at our day to day lives, you will notice that our lives are nothing but a collection of choices and decisions.

We chose to act, chose not to act, we chose to think a thought and share that thought and so on – every single moment of every single day in our lives is nothing but choices and decisions.

Our decisions are sometimes well-placed, and some other times misplaced.

But the one thing we all have in common, is that we would never make a choice or a decision without knowing that it has some sort of positive outcome one way or another.

Even criminals who make the worst of decisions, they make them for a personal gain or to make a point or a statement to the public.

Decisions become harder based on their impact and time range.

For instance, a decision to drink tea or coffee is much easier than a decision to go to war.

A decision to wear which color shirt in the morning is much simpler than a decision to prohibit the usage of cellphones in public transportations for instance.

The problem with our decisions, especially the more complex ones, is that they might be quite hard to predict their impact, not just in the near and far future but on the so many different levels of people these decisions affect.

The Problem

We fail to make the best decisions simply because we can't look into the future. There is no way for us to know what exactly is going to happen based on the decisions we make.

The best thing we can do today is speculate and predict what is expected to happen if one decision is made versus another.

But if we were to have a time-machine to allow us to leap into the future and see what our decisions in one direction or another may cause it would have been much easier for us to go back and make the right decision wholeheartedly without a doubt.

But even if we assume that time-travelling was a possibility, someone might need to go through the future in order for other versions to access that future.

In other words, the future has to exist first before we are able to travel to it – one cannot travel to a place (or time) that doesn't exist or is yet to come.

Time-travelling to make the best of decisions is a complex problem physicist have spent centuries trying to solve.

And time-travelling in and of itself requires someone to have previously made one decision or another so we can evaluate the results, learn from them and adjust.

Some others might suggest a butterfly effect methodology to try different permutations of decisions to reach the right answer – and on the contrary there's already a group of people today that would warn against breaking the space-time continuum.

These are all entertaining and exciting topics to dive into, but they are not practical for our age and time – and there's much more understanding that needs to be invested in the universe and matter to come up with a plan.

A Difference Way

A different and more practical way to leap into the future is to develop a better prediction mechanism leveraging machine learning and artificial intelligence to calculate all the different variables and permutations to determine the impact of any particular decision.

This brings back the oldest topic in computer science - which is modeling & simulation.

Modeling is the art of extracting the most problem-relevant characteristics of an entity in our world and represent that entity in a computer program or a mathematical equation to solve for the problem the modeling was intended for.

Simulation is mimicking the interactions between multiple models to visualize a result out of that interaction.

In short, modeling creates entities, and simulations makes these entities interact with one another.

Real-World Examples

In reality, if we were to make a decision about determining the maximum speed vehicles should not exceed on a highway, we need to model a vehicle entity, calculate the average speed on any particular road for the vehicles and simulate what is the maximum speed possible on that road that would not result in accidents.

If the road modeled in that problem nears an intersection, then a speed of 70 m/h would almost definitely cause issues.

A simulation of the average time it takes for a vehicle to slow down then running different permutations of scenarios where the road is crowded, ill lit, rainy or snowy and all other modeling variables that may impact the outcome of determining what the maximum speed may be.

Some decisions can be calculated instantaneously with our brains, some others cannot.

For instance, you don't need AI to tell you that deducting your payments to your subscriptions will result in the discontinuation of the services you are subscribed to.

But the question continues about a longer term and more complex problems that impacts larger range of individuals – so where do we stand today?

Where We Stand Today

Most of the prediction algorithms and machine learning investments are mainly targeting the financial sector.

There are more engineers working on the stock exchange markets to predict the next movement in a price for a particular stock than their investment in trying to understand what some decisions may result in on a much larger scale.

Or solving more critical problems like homelessness, poverty and addiction.

The main issue here is that solving homelessness isn't quite as rewarding from a financial standpoint as predicting the next price more in some stock.

Solving addiction problems, psychological issues or simply depression requires funding and has almost no material ROI from a capitalistic financial standpoint.

There are decisions that are made on much larger scales around the world that can definitely use the help of A.I to visualize the potential impact of any decisions and work on adjusting the decision to match the best results that hopefully benefits everyone.

In the software industry today, the access to machine learning algorithms, high performing cloud instances and using A.I libraries is the easiest thing an engineer could do. And the materials to learn about these subjects and direct them towards more pressing issues than increasing the capital of some company is much more important from a humanity standpoint than anything else.

Time-Travel-ish

The access to the computation power and information today puts us in a very strong position to try to tackle these problems, the problems that threaten the survival of our humankind and its evolution.

This access might not allow us to travel through time, but it might help us visualize what the future would look like if we were to take one decision or another and then eventually help us make the best decisions based on that information.