How to Become a Better Software Developer

medium.com/devtrailsio/how-to-become-a-better-software-developer-dd16072c974e

November 21, 2018

Pavels
Nov 22, 2018

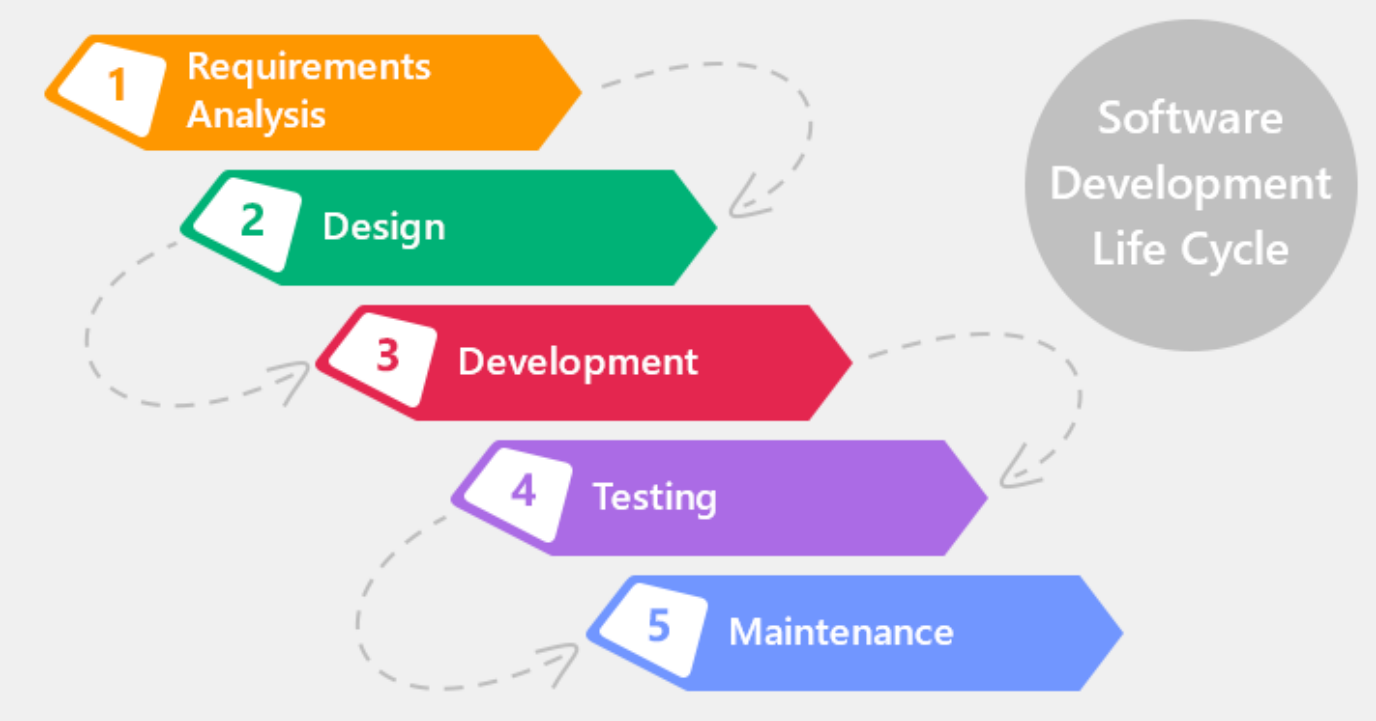
Today I would like to share some thoughts on ways a software developers can improve their professional skills and become better at their work. The topics raised here are universal and not specific to any technology stack. Most of them are not even specific to IT, for that matter. These are general advice on how to develop your personal traits, improve collaboration with colleagues and clients, and advance your career as a software developer.

Some of the things in this article are subjective and reflect my personal experience, while others have been adopted and successfully used by others.

Understand the Process End to End

A lot of developers think that software development is all about coding, and everything else is just people trying to be annoying and wasting their precious time. This cannot be further away from the truth. Before you get to code a piece of software, it undergoes a process of transformation from a vague idea into a carefully designed solution ready for implementation. And after you pushed your latest changes into Git the software is being tested, deployed, monitored, analyzed and improved on. Coding is just one of the many steps of the process.

So why does this happen? Frequently, especially when working in larger organizations, different phases of the projects are handled by different teams or even departments. It all starts with the business analysts, who gather requirements. The requirements are then handed over to the designers that produce the mockups for developers. The developers code away and give the results to the QA engineers. If everything is OK, the artifact is sent to the operations teams that deliver it to the end users. This process is treated as a set of discrete steps without any feedback. Because of the lack of communication between the departments, their representatives often don't really understand the goals of others and this leads to misunderstandings and even conflicts.



Often the process of software development is treated as a set of discrete steps with no feedback.

For many people nowadays this might sound too exaggerated. With the rise of agile methodologies, more companies move away from such a rigid approach towards smaller teams consisting of people of mixed specialty. But even then we see that people don't really try to understand the work of others. How often have you been irritated with your designers because they want you to implement a custom checkbox that is just too time-consuming? And vice-versa, received criticism, because you forgot to use the correct font.

A lot of these differences can be overcome by just paying attention to the work of others. Sit down with your designer and explain him, that implementing a custom checkbox takes a while and that there's a library that offers a different similar checkbox you could reuse. In return, learn the basics of typography and understand why choosing a correct font makes a difference. Develop the same attitudes toward managers, business analysts, QA engineers, support and marketing specialists. Quoting T. Huxley:

> Try to learn something about everything and everything about something.

By learning something from everybody, you will be able to anticipate their needs, shorten the feedback loop and enable more frequent deliveries. Plus it will earn you a lot of love and respect from everybody else.

Understand Your Client's Needs

There's one important thing that you need to understand about your customers: they don't understand most of the stuff that you're doing. Agile, functional programming or non-relational databases is all dark wizardry to them. Even the ones that closely follow your work and are genuinely interested are still mostly in the dark. This has a couple of consequences.

The face of most clients when talking to software developers.

Hiring software developers for them requires a certain degree of trust. People often tend to feel uncomfortable about having to pay a lot of money for something they don't understand. Remember last time you walked into an unfamiliar car repair service and weren't sure if you could trust them with your ride? Well, your clients have the same feeling. Except there's no car, there's just a whole bunch of abstract non-tangible concepts which are supposed to somehow materialize into products and revenue. When working with new clients it's important to earn their trust. Make sure they understand how you operate and aim to deliver results in smaller but frequent iterations. That way they can see the progress of your work, assess the intermediate results and provide their feedback.

Often clients tend to come up with their own solutions instead of sharing their problems. Since they have little idea of your capabilities, their solutions are often misjudged, under- or overambitious. Remember the old (and maybe fictional) quote by Henry Ford:

> If I had asked people what they wanted, they would have said faster horses.

Instead of going with the flow and silently implementing whatever the client wants, it's sometimes useful to invite them to take a step back and discuss the problem that they wanted to solve in the first place. When combining their domain knowledge and your technical expertise, you're are likely to arrive at a better solution.

Keep in mind, that not everybody likes having their ideas questioned and this tactic requires you to have some tact and inspire confidence in the client's eyes. You will also need to leave your comfort zone and immerse yourself in their business, to be able to understand the problem and suggest a better solution. This can be challenging if you're are working in complex industries such as finance or health care. But if you pull this off once, it's likely that next time the client will return with a more open mind.

Pick the Right Tools for the Job

> If all you have is a hammer, everything looks like a nail.

Often developers that learn only a single technology rush to apply it to every problem they encounter. Unsurprisingly, this kind of approach leads to sub-optimal results. Instead, when tackling a new problem, pause and think whether the tools at your disposal are really suitable

for this kind of work. If you have doubts, investigate a bit and come up with a list of likely superior alternatives. To make it easier, compile a list of questions and assess different options one by one. The questions can be different for each assessment, but it can go along the way of:

- What platforms or devices must it support?
- What are the non-functional requirements, such as performance or memory usage?
- Is buying a license an option, or do you need something free or open-source?
- Does the solution provide everything you need out of the box, or will you need to write something yourself?
- Do you have any other limitation, like company policies, legal considerations or a lack of specialists in your team?

Answering these questions should help you structure the options in your head and narrow them down to a shortlist of candidates.

Experiment Safely

So what happens if you none of the things you know are a particularly good fit in your case and you want to try something new? The fact that you don't experience with something doesn't automatically mean that it's out of the question. It just means that you need to consider some additional things:

- **Do you have enough time for preparation?** If the timeline of the project is not stressful, you can learn as much as possible before you begin the implementation and pick up the rest along the way. Or at least adopt the "fake it till you make it" approach and convince the client that you know what you're doing.
- **Identify the things you need to test first.** Take the "fail fast" approach and identify the crucial things that you need to evaluate before you can conclude the experiment. Having doubts about the performance of a system? Build a minimal prototype and run a load test. Uncertain about a particular library or integration with an external service? Implement that separately and then build the rest.

Keep in mind that going down this road is still risky both for you and your client, and they need to be aware of both the risks and the potential benefits. After all, a two-week investigation that might save months of work in the long run, this sounds like a pretty good deal. Even if the experiment fails, you only lose two weeks. The more trust you have with your client, the more they are likely to agree to something like this.

Build on the Shoulders of Giants

Reinventing the bicycle often leads to weird results.

IT people often have two common characteristics: we are inventive and we enjoy our work. This sounds like a good thing, but it comes with an awkward side-effect: we tend to come up with our own solutions to problems that have been solved before. So whenever we're faced with a choice of whether to use a framework, library or service or to implement it on our own, we tend to choose the latter. And this takes us on the futile journey of reinventing the wheel. Some of the common misbeliefs that lead to this are:

- **Implementing something yourself is easier than learning a 3rd party solution.** While this may be a perfectly valid reason, it's important not to oversimplify the task at hand. Often, something seems simple in the beginning but turns out to be much more difficult with progress. Eventually, you could end up spending a whole bunch of time handling bugs and edge cases that someone could have handled for you.
- **This solution does more things than I need.** Unless there are specific reasons why this is a bad thing, such as increasing the size of the resulting artifact, adding potential vulnerabilities or considerably slowing down the build, this is not usually a bad thing. You might end up needing it later. On the other hand, adding a whole library to use just one function might be an overkill.
- **We can do it better.** Although there are some successful projects that started with these words, this is not usually the case. Quality third part solutions are maintained by teams that have experience and resources devoted to solving this particular problem. To compete with them you need to be able to invest even more. Most projects have neither

the resources nor the need to do that.

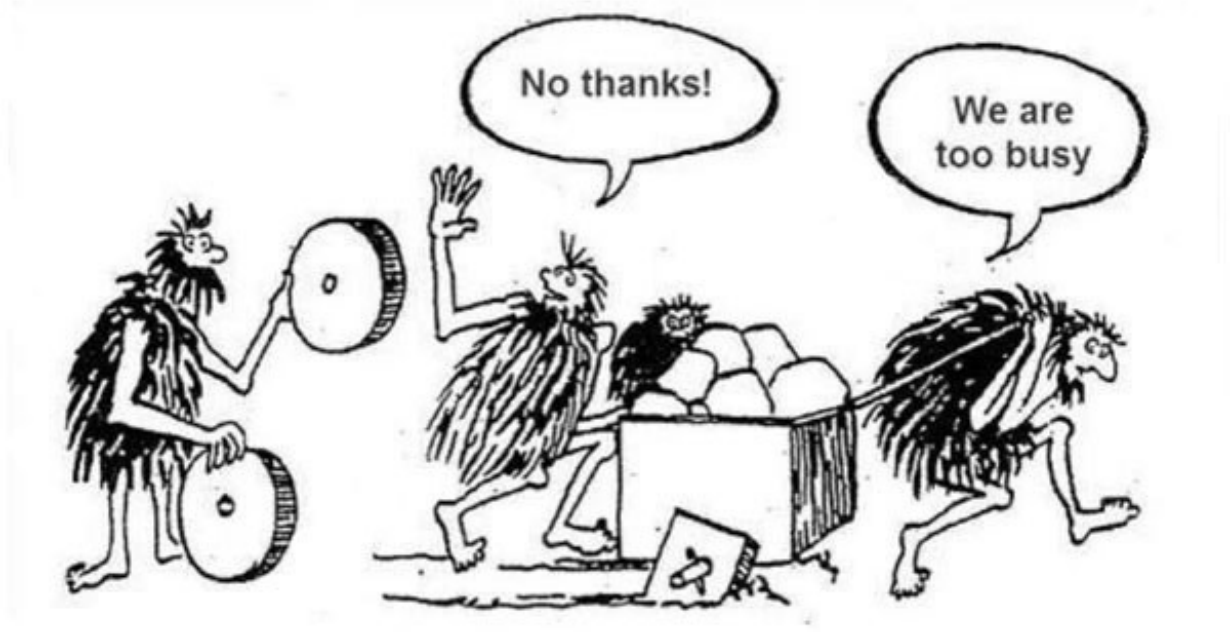
- **Code ownership and long-term maintenance will become a problem.** Some people fear that if you go with a third party solution, you risk that the project at some point might become abandoned or unusable for whatever reason. The risk of product lock-in is real, and you should consider a possible mitigation strategy. If it's an open-source project, would it be possible for you to fork it and maintain by yourself? Or if it's a proprietary project, how much would it cost to replace it? Based on these inputs you can make a conscious decision on whether it's worth the risk.

Learn Through Reimplementing

There's another side to this story. Reimplementing something yourselves is actually a great way to learn. While writing your own framework for a production project is almost always a bad idea, creating it as a learning exercise can be highly valuable. What better way to familiarize yourself with the problems that it solves by taking a crack at the same problems. Don't go too deep into the rabbit hole, a simplified crude implementation will be enough to give you an understanding of the situation.

While you're at it, don't shy away from peeking into the sources of similar projects. Studying the code of open-source projects will allow you to benefit from the experience of more seasoned developers.

Work on How You Work



Strive for improvements not just in technological aspects, but in methodological as well. Just like properly designed and optimized software, a well-established workflow will allow you to work with fewer effort and stress while producing better results. Establishing an effective and

efficient work process is not an easy task and there are numerous books and materials available on this topic. But for a start, consider the following areas for improvements:

- **Team and project management methodologies.** Since most of us work in teams, it's important to adopt a process that improves collaboration and establishes a common work rhythm for everybody. The agile movement in software development has given birth to a number of widely adopted methodologies, such as Scrum or Kanban. They help organize the overall work structure but don't cover everything. There are other methodologies that help you make estimates, prioritize issues, improve communication, etc. You'll need to identify the areas you are struggling with and look for best practices that help address your struggles.
- **Personal processes.** Like an orchestra, an effective team must have the same rhythm, but it doesn't mean that everybody must work in an identical manner. Each person has their own preferences and should work in a way that makes them more productive. For example, a lot of people don't like to be disturbed for hours when coding, but I, personally, like to work in short one-two hour bursts with breaks in between (a less strict version of the pomodoro technique). I also don't like to work at home to avoid household-related distractions and prefer to work from an office or a cafe. Find out what works for you and stick to it, but also make sure that your habits don't create problems for other team members.
- **Engineering practices**. A lot of practices lie on the border between technology and methodology and focus on improving the actual development process. For example, test-driven development and behavior-driven development help keep your code base well structured and tested. Code reviews help reduce defects in the code and also spread knowledge in the team. Continuous integration and continuous delivery ensure an easier and safer deployment process. Use these practices in combination with other organizational methodologies to achieve maximum results.

Remember, that there's no process that will work for everybody, you need to trial it in your own environment. Also, make sure that you understand the process completely and implement it correctly. Seek advice from teams that have already gone through the process and benefit from their experience. Don't neglect the software and material tools that will help you to adopt a process. Get a real Kanban board and a modern platform for continuous delivery. Adopting a new process will require effort and can even lead to a short-term loss of productivity. Give it some time and then do an evaluation of whether things have improved.

Remove obstacles

A separate thing has to be said on addressing obstacles. It's a common mistake to neglect small nuisances that might not seem important but can actually have a toxic effect on your work. Is your product designer sitting in a separate room or building? This means that it takes a bit more efforts to come over and have a conversation and some things will not be discussed. Is writing a new test difficult? Then a lot of things will not be tested.

None of these things are particularly dangerous by themselves, but they tend to pile up and cause serious consequences. And the nasty thing is, that you often don't notice them until it's already too late. Especially when there are always more serious perils to address. Remember the fable about the boiling frog and the notion of creeping normality.

Stay vigilant and fight these things at their roots before they get to you.

Focus on the Fundamentals

Fundamental concepts are the building blocks of your career.

IT is an extremely fast-paced industry. Every week new tools are released into the market. I've already mentioned the infamous "JavaScript fatigue" syndrome in my previous post. A lot of developers have been stressed and felt forced to re-evaluate their JS tech stack with each new project and that drove them nuts. Indeed, always being on the edge can be challenging, but there are a couple of ideas that can make it easier.

First of all, focus on the fundamentals. Even though new technologies appear quite frequently, new fundamental concepts are much more seldom. When learning something new, make sure you understand the underlying ideas that lead to this implementation. Chances are, these ideas are used in other projects as well, and once you encounter something similar, it will be easier for you to get a grasp of it. For example, modern JavaScript UI frameworks are based on

components, and once you understand how to structure a component-oriented application using React, can use this experience when working Angular. In a similar manner ideas of Redux found their way into Angular, and reactive state management from Angular was implemented for React as MobX.

Take some time to familiarize yourself with the classical books on the topics of common patterns in software development such as “Enterprise Integration Patterns” by Gregor Hohpe and Bobby Woolf, the famous “Design Patterns: Elements of Reusable Object-Oriented Software” by the Gang of Four or different works of Martin Fowler. Although some of the things described in books may be outdated, you can use them to learn how the patterns evolved till today.

Secondly, don’t rush to learn about every new thing out there. I understand that it’s tempting to follow every new thing that appears on Hacker News, but a lot of these things are just noise. Rather keep an eye out for things that have been circling in the community for some time now and have matured beyond the hype of initial discussions. Don’t give into FOMO. If you pay at least some attention to what’s going on, no important thing will pass unnoticed.

Bonus Tips

We've already talked about a lot in this article, but there are a few other points I would like to highlight before we wrap up. These few tips are focused more on your personal traits rather than professional, but I still believe that they have a high impact on your work life.

Often people think that hoarding valuable knowledge will make them indispensable. Having people like this in your team exposes you to the "bus factor" and can put you in a tough spot if such a person were to leave the project. If you are one of these people, consider that in addition to making you indispensable, your expertise also makes you unpromotable and "unvacationable". You might miss out on other career opportunities in your organization because you are tied up in this role. Instead, share the knowledge with your colleagues, if possible delegate part of your work to them and use this collaboration to build even greater things on top of their work.

Don't blame yourself or others

I remember a long time ago we had an incident in one project that was by my mistake. We've managed to recover from the incident quite quickly and I remember the client telling me:

> You don't judge a team by how they perform when everything goes according to plan, but by how they operate when the shit hits the fan.

No matter how good you are, sometimes things will go wrong and in such moments it's important to be able to keep your cool and handle the situation with dignity and mutual respect. After the fire is put out, don't focus on finding the scapegoat. This won't help you avoid mistakes in the future, but will spear fear and doubt across the company. Instead, come together with the affected parties and do a common post-mortem. Focus on the things that lead to the problem, figure out why it happened and brainstorm on what you can improve your system or workflow to avoid this problem in the future.

Don't be an asshole

The developer community is a funny thing. On one side we see a lot of driven open-minded people that contribute to the community by working on open-source projects, giving speeches at conferences or writing articles. On the other side, we encounter people that troll new ideas, disrespect newcomers and demonstrate rude behavior to everyone around them. Don't be one of those people. Be nice and spread the love.

A lot of professional advice can be summed up with just four words.

Wrapping it up

The best thing about our work is that it doesn't have a limit. There are always new roads to travel and dragons to slay. Whether you're just at the beginning of your journey or an experienced professional, keep these things in mind. They should help you find your way and become a better developer with each step taken.

Do you have different advice to share with others? Feel free to post it in the comments and start a discussion!

Are you interested in learning web development or further improving your skills? Check devtrails.io for a collection of helpful guides to help you figure out your way around web development.