Photo by NESA by Makers on Unsplash

FEATURED STORIES

Notes to Myself on Software Engineering

A laundry list of personal reminders

François Chollet Follow
Sep 8, 2018 · 7 min read ★

On the Development Process

1. Code isn't just meant to be executed. Code is also a means of communication across a team, a way to describe to others the solution to a problem. Readable code is not a nice-to-have, it is a fundamental part of what writing code is about. This involves factoring code clearly, picking self-explanatory variable names, and inserting comments to describe anything that's implicit.

2. Ask not what your pull request can do for your next promotion, ask what your pull request can do for your users and your community. Avoid "conspicuous contribution" at all cost. Let no feature be added if it isn't clearly helping with the purpose of your product.

3. Taste applies to code, too. Taste is a constraint-satisfaction process regularized by a desire for simplicity. Keep a bias toward simplicity.

4. It's okay to say no — just because someone asks for a feature doesn't mean you should do it. Every feature has a cost that goes beyond the initial implementation: maintenance cost, documentation cost, and cognitive cost for your users. Always ask: Should we really do this? Often, the answer is simply no.

5. When you say yes to a request for supporting a new use case, remember that literally adding what the user requested is often not the optimal choice. Users are focused on their own specific use case, and you must counter this with a holistic and principled vision of the whole project. Often, the right answer is to extend an existing feature.

6. Invest in continuous integration and aim for full unit test coverage. Make sure you are in an environment where you can code with confidence; if that isn't the case, start by focusing on building the right infrastructure.

7. It's okay not to plan everything in advance. Try things and see how they turn out. Revert incorrect choices early. Make sure you create an environment where that is possible.

8. Good software makes hard things easy. Just because a problem looks difficult at first doesn't mean the solution will have to be complex or hard to use. Too often, engineers go with reflex solutions that introduce undesirable complexity (*Let's use ML! Let's build an app! Let's add blockchain!*) in situations where a far easier, though maybe less obvious, alternative is available. Before you write any code, make sure your solution of choice cannot be made any simpler. Approach everything from first principles.

9. Avoid implicit rules. Implicit rules that you find yourself developing should always be made explicit and shared with others or automated. Whenever you find yourself coming up with a recurring, quasi-algorithmic workflow, you should seek to formalize it into a documented process, so that other team members will benefit from the experience. In addition, you should seek to automate in software any part of such a workflow that can be automated (e.g., correctness checks).

10. The total impact of your choices should be taken into account in the design process, not just the bits you want to focus on — such as revenue or growth. Beyond the metrics you are monitoring, what total impact does your software have on its users, on the world? Are there undesirable side effects that outweigh the value proposition? What can you do to address them while preserving the software's usefulness?

Design for ethics. Bake your values into your creations.

On API Design

1. Your API has users, thus it has a user experience. In every decision you make, always keep the user in mind. Have empathy for your users, whether they are beginners or experienced developers.

2. Always seek to minimize the cognitive load imposed on your users in the course of using your API. Automate what can be automated, minimize the actions and choices needed from the user, don't expose options that are unimportant, design simple and consistent workflows that reflect simple and consistent mental models.

3. Simple things should be simple, complex things should be possible. Don't increase the cognitive load of common use cases for the sake of niche use cases, even minimally.

4. If the cognitive load of a workflow is sufficiently low, it should be possible for a user to go through it from memory (without looking up a tutorial or documentation) after having done it once or twice.

5. Seek to have an API that matches the mental models of domain experts and practitioners. Someone who has domain experience, but no experience with your API, should be able to intuitively understand your API using minimal documentation, mostly just by looking at a couple of code examples and seeing what

objects are available and what their signatures are.

6. The meaning of an argument should be understandable without having any context about the underlying implementation. Arguments that have to be specified by users should relate to the mental models that the users have about the problem, not to implementation details in your code. An API is all about the problem it solves, not about how the software works in the background.

7. The most powerful mental models are modular and hierarchical: simple at a high level, yet precise as you need to go into details. In the same way, a good API is modular and hierarchical: easy to approach, yet expressive. There is a balance to strike between having complex signatures on fewer objects, and having more objects with simpler signatures. A good API has a reasonable number of objects, with reasonably simple signatures.

8. Your API is inevitably a reflection of your implementation choices, in particular your choice of data structures. To achieve an intuitive API, you must choose data structures that naturally fit the domain at hand — that match the mental models of domain experts.

9. Deliberately design end-to-end workflows, not a set of atomic features. Most developers approach API design by asking: *What capabilities should be available? Let's have configuration options for them.* Instead, ask: *What are the use cases for this tool? For each use case, what is the optimal sequence of user actions? What's the easiest API that could support this workflow?* Atomic options in your API should answer a clear need that arises in a high-level workflow — they should not be added "because someone might need it."

10. Error messages, and in general any feedback provided to a user in the course of interacting with your API, is part of the API. Interactivity and feedback are integral to the user experience. Design your API's error messages deliberately.

11. Because code is communication, naming matters — whether naming a project or a variable. Names reflect how you think about a problem. Avoid overly generic names (*x, variable, parameter*), avoid *OverlyLongAndSpecificNamingPatterns*, avoid terms that can create unnecessary friction (*master, slave*), and make sure you are consistent in your naming choices. Naming consistency means both internal naming consistency (don't call "dim" what is called "axis" in other places) and consistency with established conventions for the problem domain. Before settling on a name, make sure to look up existing names used by domain experts (or other APIs).

12. Documentation is central to the user experience of your API. It is not an add-on. Invest in high-quality documentation; you will see higher returns than investing in more features.

13. Show, don't tell: Your documentation should not talk about how the software works, it should show how to use it. Show code examples for end-to-end workflows; show code examples for each and every common use case and key feature of your API.

Productivity boils down to high-velocity decision-making and a bias for action.

On Software Careers

1. Career progress is not how many people you manage, it is how much of an impact you make: the differential between a world with and without your work.

2. Software development is teamwork; it is about relationships as much as it is about technical ability. Be a good teammate. As you go on your way, stay in touch with people.

3. Technology is never neutral. If your work has any impact on the world, then this impact has a moral direction. The seemingly innocuous technical choices we make in software products modulate the terms of access to technology, its usage incentives, who will benefit, and who will suffer. Technical choices are also ethical choices. Thus, always be deliberate and explicit about the values you want your choices to support. Design for ethics. Bake your values into your creations. Never think, *I'm just building the capability; that in itself is neutral.* It is not because the way you build it determines how it will get used.

4. Self-direction — agency over your work and your circumstances — is the key to life satisfaction. Make sure you grant sufficient self-direction to the people around you, and make sure your career choices result in greater agency for yourself.

5. Build what the world needs — not just what you wish you had. Too often, technologists live rarefied lives and focus on products catering to their own specific needs. Seek opportunities to broaden your life experience, which will give you better visibility into what the world needs.

6. When making any choice with long-term repercussions, place your values above short-term self-interest and passing emotions — such as greed or fear. Know what

your values are, and let them guide you.

7. When we find ourselves in a conflict, it's a good idea to pause to acknowledge our shared values and our shared goals, and remind ourselves that we are, almost certainly, on the same side.

8. Productivity boils down to high-velocity decision-making and a bias for action. This requires *a*) good intuition, which comes from experience, so as to make generally correct decisions given partial information, *b*) a keen awareness of when to move more carefully and wait for more information, because the cost of an incorrect decision would be greater than cost of the delay. The optimal velocity/quality decision-making tradeoff can vary greatly in different environments.

9. Making decisions faster means you make more decisions over the course of your career, which will give you stronger intuition about the correctness of available options. Experience is key to productivity, and greater productivity will provide you with more experience: a virtuous cycle.

10. In situations where you are aware that your intuition is lacking, adhere to abstract principles. Build up lists of tried-and-true principles throughout your career. Principles are formalized intuition that generalize to a broader range of situations than raw pattern recognition (which requires direct and extensive experience of similar situations).

Software Development Work Programming Self Productivity