Member-only story

10 Habits of Great Software Engineers

Great developers tend to have a lot of little habits that they don't teach in Computer Science courses or web development bootcamps, and these 10 habits will make you stand out.

Dr. Derek Austin 🥳 · Follow
Published in Career Programming · 11 min read · Sep 11, 2024

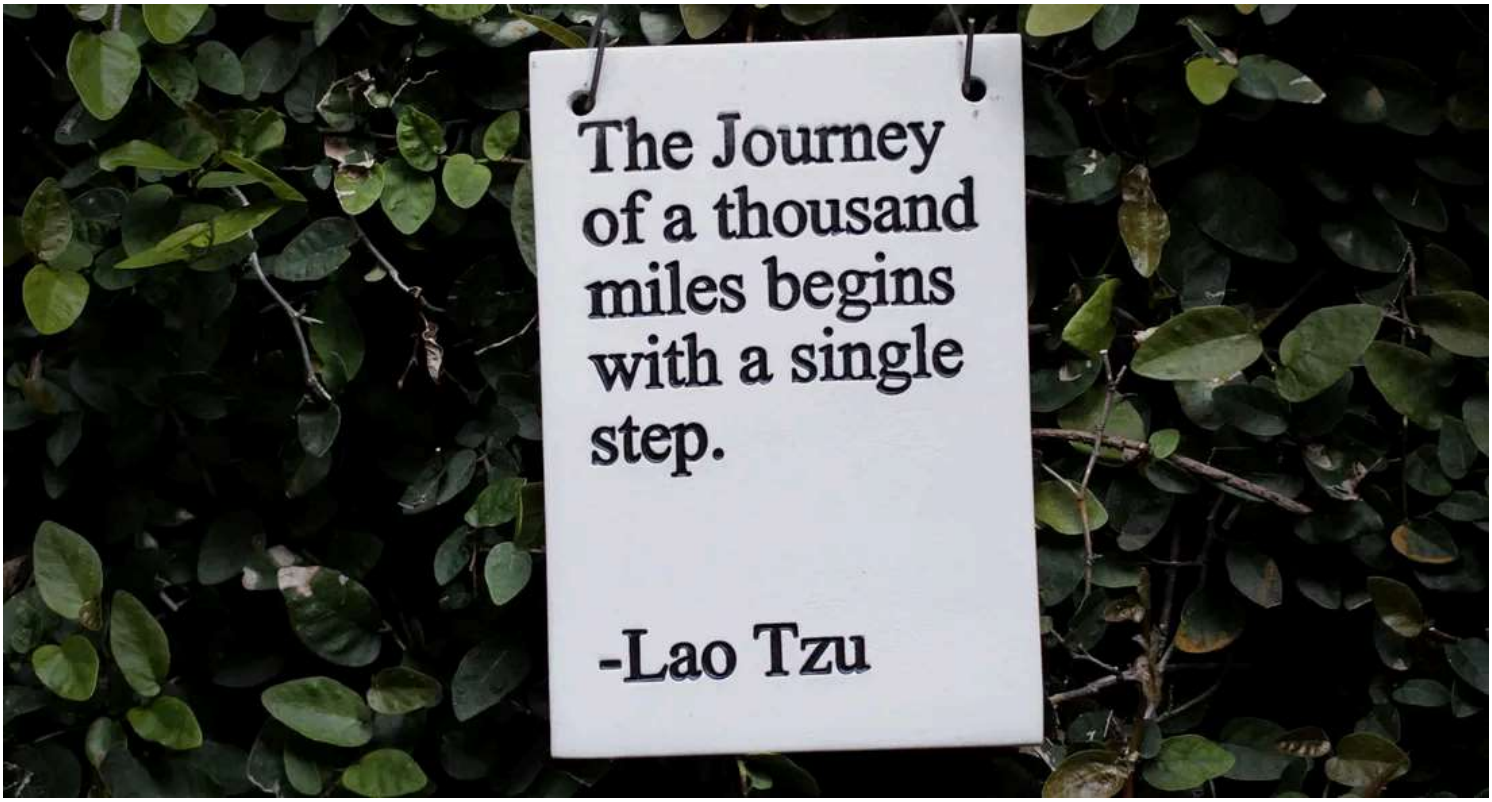


Photo by Hester Qiang on Unsplash

I've been developing professionally since 2004, and there is a lot about working on a software engineering team that I didn't learn until recently.

Over the last 5 years, I've had the privilege of working as a lead frontend engineer and hiring manager for a lot of different teams, big and small.

That's why I wanted to put together this article and share 10 little habits that have made the great developers on my teams stand out.

Habit 1 — Create ready-to-work tickets.

Every professional software engineer has to write up tickets, or work items, even when you're supported by a full-time product manager.

Usually, you (the engineer) know what the bug or feature request is, so it's tempting to write a 1-line ticket: just the title, with nothing else.

Actual tickets need something called "acceptance criteria," which is another fancy term, this time meaning "the definition of done."

You and your team need to agree in advance what the QA (testing) process is going to be for the ticket. That's the acceptance criteria (or AC, for short).

One of the best ways to succeed as a software engineer at any level is to write up great tickets that are ready to work and ready to test.

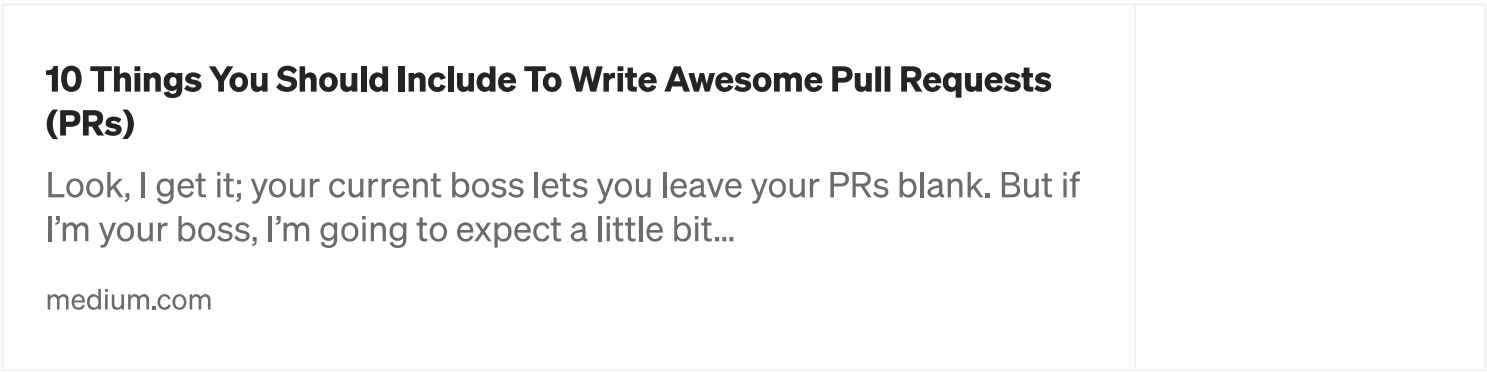
Great tickets typically contain a user story (*why* you're coding, from the user's perspective) and some testing instructions, though depending on the work you may have a lot more additional details included in the ticket.

While it's typically the product manager's job to write up tickets, we always find bugs while coding, and so we should also write up tickets as needed.

And the best way to get on your product manager's good side as an engineer is to write fantastic tickets, without needing to be asked.

Habit 2 — Author ready-to-review PRs.

As a developer on a software team, you typically turn in your work by opening pull requests (PRs), requests to merge your code in.



The first thing any PR should include is a link back to the ticket being completed, even if that means just naming the PR the name of the ticket.

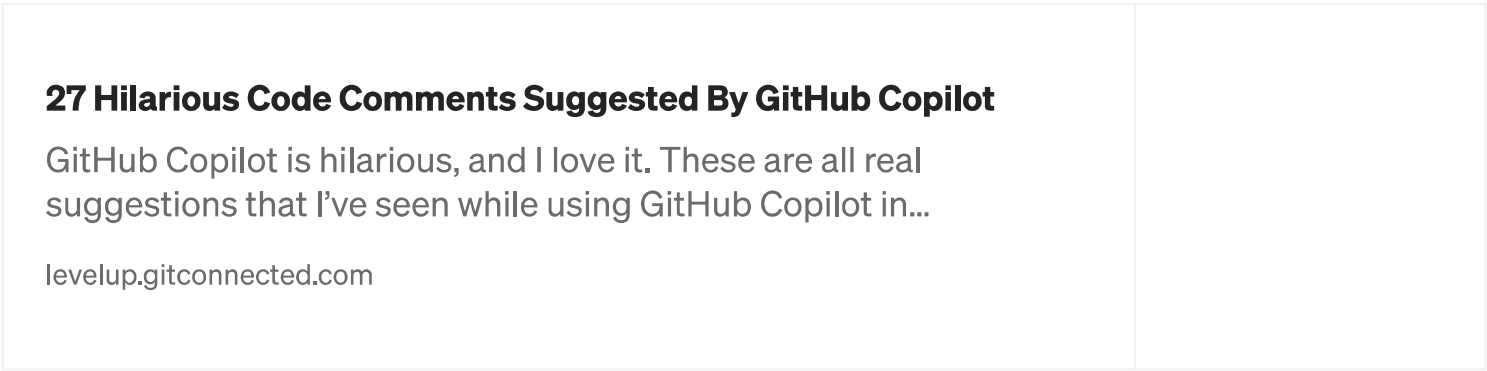
You can also include some additional details, like:

- What was the old behavior, and what is the new behavior?
- Do I need to actually test anything in the codebase?
- Is there anything else I need to know about the code?

It only takes a couple of minutes to describe what you're turning in as part of your PR, and it's a big help to whoever's going to review your code.

Habit 3 — Write lots of code comments.

I write as many lines of code and as lines of code comments, because I think code comments are extremely important for every codebase.



Code comments are not a waste of time, and in fact they're the single most important factor in my experience for the maintainability of a codebase.

While I definitely encourage you to write "self-documenting" code where *how* the code works is clear, *why* it has to work that way is often less clear.

Let's take a common scenario as an example.

You're a new hire with professional experience in the tech stack being used at your new company. You're assigned to fix a simple, easy-to-find bug.

You get to the relevant section of code, and you don't have any code comments at all. You focus and read the algorithm line by line.

Nearly an hour later, you finally understand how the code works! Now you can start fixing the bug! That's when you should write code comments.

All you need to do to write great code comments is explain *why* there was a bug, which is the same as explaining *why* the code is written the way it is.

Habit 4 — Make atomic commits.

Speaking of technical documentation, an underrated approach to coding is to commit small changes, frequently, by using "atomic commits."

An atomic commit is a single, indivisible code change, just like how an atom was an indivisible piece of matter before the discovery of quarks.

Think about it this way: how often do you save an essay? As often as possible, preferably with an autosave feature. Maybe every 10 minutes?

You should be saving your code that often, just to be safe. Of course, it's not typically necessary, but you'll be happy you saved when you do need it.

For example, in a recent refactor ticket I worked, I made **36 atomic commits** to change **34 files**, which took me **8 hours** from start to finish.

The best thing I've found from making atomic commits is that it's exceptionally easy for me to revert a group of changes.

Instead of what I used to do — save my work once a day, or maybe twice — I'm able to fully implement a solution, saving several commits as I go.

If I want to try another solution, I'm able to revert *just those commits*, saving the first solution (via Git) in case the second solution doesn't work.

5 — Use semantic commit messages.

Speaking of Git commit messages, I'm a huge fan of semantic commit messages, meaning each commit is labeled with a type and a scope.

I find that having to write 20+ commit messages a day clarifies my thinking, making sure I understand exactly what I'm trying to do.

Even better, writing semantic commit messages for easy tasks (like `docs(backend): improve code comments in api.tsx`) helps me build momentum for harder tasks (`feat(backend): implement new endpoint`).

Semantic Commit Messages

See how a minor change to your commit message style can make you a better programmer.

Format: `<type>(<scope>): <subject>`

`<scope>` is optional

Example

```
feat: add hat wobble
^--^  ^------------^
|     |
|     +-> Summary in present tense.
|
+-------> Type: chore, docs, feat, fix, refactor, style, or test.
```

More Examples:

- `feat` : (new feature for the user, not a new feature for build script)
- `fix` : (bug fix for the user, not a fix to a build script)
- `docs` : (changes to the documentation)
- `style` : (formatting, missing semi colons, etc; no production code change)
- `refactor` : (refactoring production code, eg. renaming a variable)
- `test` : (adding missing tests, refactoring tests; no production code change)
- `chore` : (updating grunt tasks etc; no production code change)

References:

- https://www.conventionalcommits.org/
- https://seesparkbox.com/foundry/semantic_commit_messages
- http://karma-runner.github.io/1.0/dev/git-commit-msg.html

It takes me 30 seconds to make an atomic commit with a 10-word commit message. Writing `WIP` or `chore: fix bug` is fine when you're a solo developer, but those commits aren't useful for your team's code review.

Let's say I have to review your pull request, but you're already done with work (East Coast) while I'm still here for 3 more hours (West Coast).

That's another time when semantic commits shine, because I want to understand *why the code changed*, but you're not at work for me to ask.

Code comments, atomic commits, and semantic commit messages are also excellent tips to make your portfolio projects "professional-grade."

And, while we're on the topic of portfolio projects, please deploy your projects to a host (like Vercel) and write useful READMEs. Thanks!

6 — DRY (Don't Repeat Yourself).

I don't love pithy coding acronyms like "DRY" (Don't Repeat Yourself), but this particular acronym is a great one for maintainability.

You've probably seen "copypasta code," meaning code that's been copy-pasted to different parts of the codebase with only minor changes.

Don't Choose the Wrong Abstraction When You DRY Your Codebase

"Don't Repeat Yourself" (DRY) is a truism in software engineering, but it can cause headaches if you choose the wrong...

medium.com

DRY just means you should be in the habit of refactoring, or rewriting your code to be easier for the whole team to maintain over the long term.

In particular, you want to refactor duplicate code, especially anything duplicated 3+ times in a codebase.

If you never refactor duplicated code, your codebase could be on its way to becoming an unmaintainable nightmare, so DRY.

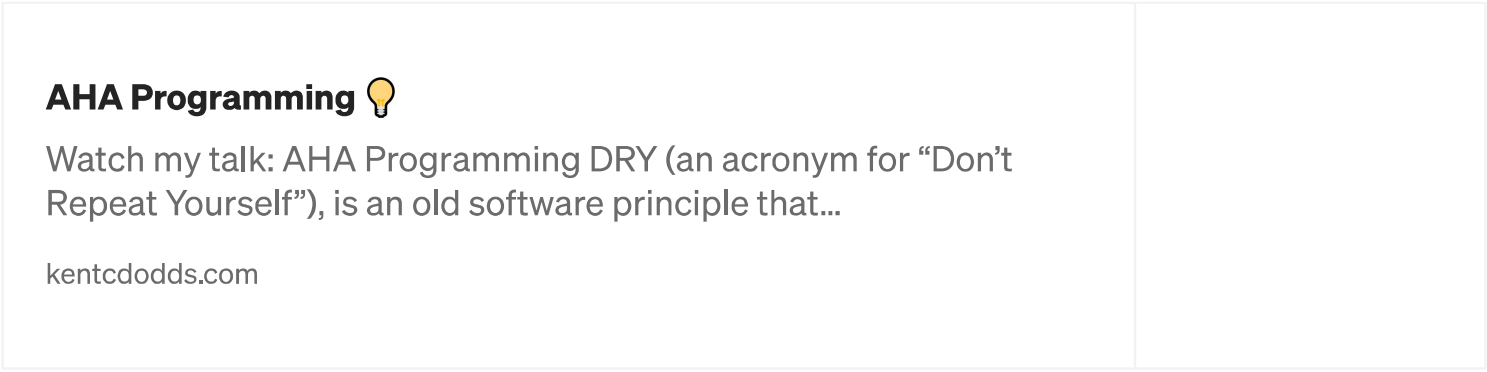
7 — WET (Write Everything Twice).

The flip side of DRY is WET (Write Everything Twice). Get it? Haha. 😁 Anyway, WET means that it's faster to copy-paste once than to refactor.

The idea with WET is if you're working on a similar feature for the 2nd time, then you want to take your time and do some thinking.

Usually by the time you're copy-pasting the same line for the 3rd time, you have a pretty good sense of what it would look like to refactor.

However, as Kent C. Dodds says, you need to AHA (Avoid Hasty Abstractions) — you don't want to have to refactor again and again.



Not everything has to be abstracted away into a neat, reusable function in order to be maintainable. I find it's best to start with WET: write it twice.

Once I've written out similar code 2 or 3 different times, I'll be able to identify a useful abstraction for it.

However, if I try to guess at that abstraction on my first attempt at writing the code, I'm often completely wrong.

That ends up taking longer than if I had just avoided hasty abstractions (AHA) by writing everything twice (WET) before I refactor.

8 — Write readable code.

I'm a huge fan of readable code as well as simple code because readable, simple code is much easier to maintain over months and years than hard-to-read, complex code that could have been simplified but never was.

What is readable code, you say? It's using things like `index` instead of `i`, or `roundNumber()` instead of `r()` for a JavaScript round number function.

I'd include always passing objects as function parameters as something that helps make my code more readable as well. Readability is important!

Of course, readability doesn't replace writing code comments, nor does it replace making atomic commits with semantic messages.

Writing readable code is just another way of promoting maintainability in your codebase, whether you or another dev are the one to maintain it.

9 — Fix ESLint warnings and errors.

Like many of you, I have a love-hate relationship with code linting, that tool in your code editor that's always pointing out potential code problems.

I love catching bugs before they make it to production, but I hate having to fight the endless squiggly lines about things that *aren't* bugs.

I get it, some ESLint rules suck.

(I'm looking at you, `"react/jsx-no-bind": "off"` — I'm an adult, and I want to pass destructured functions as props.)

But, if you find yourself starting to type `// eslint-disable-next-line`, are you 100% sure you don't need that rule?

Of course, every team is different, but I've found that my productivity is best when I turn off the rule "at the source," in the `.eslintrc.js` configuration file, along with a code comment explaining *why*.

If that's not an option, or you like the rule in general but not this time, then I suggest adding another code comment to `// eslint-disable-next-line`.

On the line before that, add a quick, one-line code comment explaining exactly that: *why* you have to turn the rule off this one time.

Since linting isn't a built-in feature to GitHub's pull request code review process, it's easy to miss linting issues before the code gets merged.

Fixing your ESLint warnings and errors (when possible) and clearly commenting them out (when not) is also great for maintainability.

10 — Meet the acceptance criteria.

As a developer, your job is typically to deliver a pull request (PR) to close a ticket with certain acceptance criteria (AC), a certain "definition of done."

Your code should meet that AC. Your code should work, even if you don't have time to write automated tests (e.g. using React Testing Library).

Unfortunately, sometimes developers to send in a ticket where either they didn't meet all of the AC correctly or they changed some of them.

It's no problem at all if you need to change the AC, just communicate it to your team (asynchronously, such as using Slack, Teams, JIRA, or Linear).

But it's also easy to forget to develop a bad habit of forgetting to communicate, resulting in people always thinking your work is "wrong."

Sure, stuff happens, but having to answer questions from your product manager or QA team about your code should be rare, not commonplace.

The more you ensure that your code meets all AC, including any AC that have changed since the ticket was written, the happier your team will be!

Conclusion: 10 Habits of Great Software Engineers

And there you have it! These are 10 habits that I've seen firsthand over the last 5 years, things that make certain developers a pleasure to work with:

1. Create ready-to-work tickets.
2. Author ready-to-review PRs.
3. Write lots of code comments.

4. Make atomic commits.
5. Use semantic commit messages.
6. DRY (Don't Repeat Yourself).
7. WET (Write Everything Twice).
8. Write readable code.
9. Fix ESLint warnings and errors.
10. Meet the acceptance criteria.

Of course, great developers have many other habits that they've developed over the course of their careers, so this isn't an exhaustive list.

These are just some of the most common habits that have come up during real code review or team retros ("retrospectives").

They're also habits that I've trained my own engineering employees on as a mentor and engineering manager.

I hope that you find these habits useful as you continue your own journey of continuous improvement ("kaizen") as a software engineer.

And, if I missed any habits of great developers, I hope you'll respond to this article to let me know!

Happy coding! 

Further Reading: More Great Developer Advice