Faire's engineering handbook

Daniele Perito · Follow
Published in The Craft · 7 min read · Feb 11, 2020

The purpose of this document is to share, promote and discuss best practices among engineers.

This document summarizes and distills the knowledge of what processes, tools and techniques have or haven't worked for us here at Faire. Some of the tenets and recommendations are subjective. In these cases, the purpose of the document is to promote convention and consistency, as these things help with collaboration.

The 15 Minute Principle

Before interrupting someone else with a question, spend 15 minutes researching the question (unless the world is on fire).

The same concept has been expressed in a lot of different ways. For example, "RTFM" is a popular acronym that expresses a related sentiment (a lot less kindly). The 15 minute principle provides constructive advice along the same lines.

There are several advantages stemming from this principle:

1. In most cases, after 15 minutes of research, you should be able to answer the question yourself, without interrupting the flow of someone else's work
2. Even if you don't find the answer, you'll learn a bit more about how the system works by searching for an answer yourself. That increases the shared knowledge of the system and it reduces the chances that next time

you'll find the need to ask a question

3. It incentivizes you to learn the tools and dig into other people's code. Being able to follow code references with your favorite IDE goes a very long way in being self sufficient

Promote Cleanliness

Naming inconsistencies, commented out code, unhelpful commit messages have lasting negative effects on your code base.

A general theme of this handbook is discussing ways to fight chaos. Code bases are large and complex entities that naturally devolve into a chaotic mess unless work is actively put to prevent that. That's a universal principle that applies to everything.

Tests, code reviews, post-mortems are all tools we use to fight entropy and chaos in large codebases.

Promoting cleanliness is another way to fight chaos. Unclean, inconsistent code begets more unclean and inconsistent code that devolves into chaos.

Investigate Issues

It's natural to try to explain away small issues that we can't quite figure out immediately. Typically, people will say things like "it's a small race condition" or "it only happens in the staging environment" without fully understanding how an issue happened. That's wrong.

Small issues always compound into large issues. You need to actively fight entropy and chaos. Whenever something doesn't make sense, dig into it. More often than not you'll find that the issue actually is part of a larger issue that needs to be fixed or at the very least explained.

Code Reviews: Be Kind, Thorough and Aware of the Urgency

1. Be kind in code reviews. A practical advice here is to phrase comments as suggestions in the form of a question (e.g., How about changing thisName to thatName?)
2. Qualify how strongly you feel about something and whether you think that something is a "nit" vs a merge blocker
3. If you consistently disagree with someone on some pattern, have an offline discussion with them to clarify your respective opinions. PR comments aren't the best place for really long discussions
4. Be thorough, the responsibility of shipped code lies both on the writer and the reviewer
5. Understand how urgent a code review and whether your teammate is blocked on it. If you know a teammate is blocked on your code review, review urgently. There are probably many people who are waiting on that code review (your fellow engineer, a PM waiting to look at the feature in staging, your customers, etc.).
6. A corollary of the rule above is, ask how urgent a review is if you are not sure
7. Accept criticism as a natural part of the process

A corollary to this principle is, review your own code before asking for

someone else's review. The reviewer's time should not be spent finding small typos or debug statements left lingering. In general, be mindful of other people's time just as much as you are mindful of your own time.

Tests

Tests are one of the most discussed topics in engineering. There is a huge number of articles and books describing what to test, how to test, promoting writing tests, test frameworks, etc. It's hard to add to that huge body of knowledge.

I suspect that part of the reason why we write so much about tests is that tests don't seem to be strictly necessary to ship code. Our lazy selves make up all kinds of excuses to tell us that we don't *really* need to write tests. That couldn't be further from the truth, tests are absolutely essential to maintaining code clean.

A few short things to summarize:

1. Tests are A Good Thing™
2. If you don't see tests while reviewing a PR, ask for them
3. Test basically always pay themselves off, often immediately. There is a false dichotomy that you can either ship code fast or you can ship well tested code. More often than not tests pay themselves off in the natural course of writing a PR.

Optimize your code for readability

This is another subject that has been discussed at length. A few key takeaways:

1. Code is read a lot more than it is written. Therefore you should write code that is easy to read.
2. Use descriptive names. Naming is one of the hardest things to get right in programming. Avoid abbreviations as much as possible (a variable called current is a lot better than a variable called curr, the latter imposes a tiny cognitive load every time it is read)
3. Follow the style guide for consistency. When reviewing code make sure it meets the style guide suggestions.
4. Avoid long methods (typically longer than 20–50 lines of code). Long method names usually indicate that the method is doing more than it should. Break it up into smaller and simpler methods with a single utility.
5. Add comments to your code. With one major caveat, comments cannot help make sense of spaghetti code. A typical example are very long methods with comments every couple of lines. A better approach in those cases is to refactor the code into smaller, self-describing method calls

Use Post-mortems to identify the root cause of past issues

If you haven't read it yet, we highly recommend you read the *Checklist Manifesto*. The general idea of the book is how it's possible to minimize faults in a system by learning from past mistakes and putting together [in software engineering, automated] processes to deal with those common mistakes. The most widely cited example of this is aviation safety

The most widely cited example of this is aviation safety.

Since the 70s, the number of deaths from crashes has been in steady decline even though the number of commercial passengers has increased dramatically. This huge success is the direct result of relentless investigation of past issues and adopting the learning from them.

Whenever there is a major bug or outage, hold a post-mortem meeting detailing the following:

1. Time issue started
2. Time issue identified
3. Time issue resolved
4. Number of customers impacted
5. Root of the problem
6. Action items to prevent the issue from happening in the future

#6 is perhaps the most important part of a post-mortem. Typically, you'll want to add better metrics, tighter controls for certain common mistakes, more automation.

One thing to be careful of is avoiding adding toil when learning from post-mortems. As much as possible, you should try to prevent future mistakes with automation. Sometimes manual process is necessary, but always think about ways to automate the learnings.

Metrics and Alerting

In general, you are what you measure. This is a widely shared tenet that applies to more than engineering.

Make sure you have at the very least performance metrics and alerting set up. The existence of performance metrics will, in a very real sense, make your application faster. Alerting, when correctly applied, will make your application less bug ridden.

Other metrics you might want to keep an eye on are: test speed, test coverage and compilation speed.

Learn the tools

The tools you use are as important for your productivity as the language you choose or the technology you adopt. They can be a huge help, or an hindrance to your day to day happiness and productivity.

In general, you want to use an IDE that deeply understands the language you are using. The IDE will be able to discover a lot of issues for you.

Another important thing to learn is using a debugger. A lot of inexperienced engineers develop with "debug prints". Debug prints are a classic example of toil (covered below) and should be avoided in a lot of circumstances.

A rule of thumb is that if you often find yourself having to deal with issues that stem from typos, you aren't using the right tools or you aren't using your tools as they are meant to be used. Find some time to shadow or pair with

someone who is more familiar with the tools you use and see how they use them.

If It Ain't Broke, Don't Refactor It

Every line of code in every codebase could be likely rewritten in a flashier or more elegant way. While we all love writing code, we should be always be mindful of the fact that we aren't writing code for its own beauty, we are writing code to help our customers. Large refactors are usually to be avoided, while smaller refactors and cleanups during the course of regular programming tasks are preferred and encouraged.

Minimize Toil

Toil is very well described in this section of the Google SRE's Handbook. We suggest you spend some time reading that handbook.

From that book: "Toil is the kind of work [tied to running a production service] that tends to be manual, repetitive, automatable, tactical, devoid of enduring value, and that scales linearly as a service grows."

A software engineer's time should be spent engineering solutions to problems. You should minimize toil and reduce it to at most 10–20% of your time (unless you are a production engineer). A lot of toil can be eliminated by better tools.

Appendix — Recommended Reading

1. *Google SRE Handbook* (Free)
2. *The Checklist Manifesto*
3. *Clean Code*

Software Development | Engineering | Engineering Mangement

207

Written by Daniele Perito

55 Followers · Editor for The Craft

Follow

More from Daniele Perito and The Craft

Daniele Perito in The Craft

How data and machine learning shape Faire's marketplace

By: Daniele Perito, Chen Peng

Oct 28, 2021 180 3

Luke Bjerring in The Craft

Boosting performance: Faire's transition to NextJS

How Faire navigated the migration from a mature single-page application to the serve...

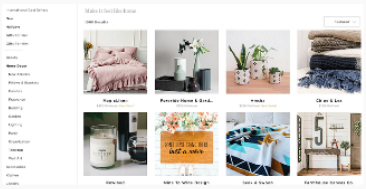
Jul 31 401 1

Luke Bjerring in The Craft

Automated code reviews with LLMs

How Faire implemented an LLM-powered automated review pipeline to boost...

4d ago 158 1



Daniele Perito

Ranking to maximize economics

One of the most important jobs at Faire is helping retailers find products that'll sell in...

Feb 11, 2020 2

See all from Daniele Perito | See all from The Craft

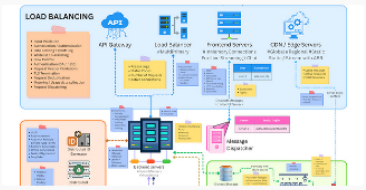
Recommended from Medium

Alexander Nguyen in Level Up Coding

The resume that got a software engineer a $300,000 job at Google.

1-page. Well-formatted.

Jun 1 18.6K 302



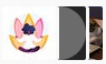
Love Shar... in ByteByteGo System Design Allian...

System Design Blueprint: The Ultimate Guide

Developing a robust, scalable, and efficient system can be daunting. However,...

Sep 17, 2023 8.6K 60

Lists



Stories to Help You Grow as a Software Developer

19 stories · 1293 saves

General Coding Knowledge

20 stories · 1502 saves

Leadership

54 stories · 407 saves

Coding & Development

11 stories · 754 saves

Dr Stuart Woolley in CodeX

Is Culture Fit Just A Cover For Covert Discrimination?

Interview processes are becoming increasingly dehumanised, as a result...

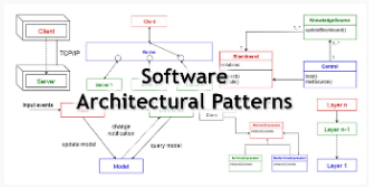
Dec 21, 2023 352 15

Bella L in ILLUMINATION

Can You Pass This Apple-Orange Interview At Apple 🍎?

The iPhone Company's Interview Question

Mar 14 7.6K 209



Vijini Mallawaarachchi in Towards Data Science

10 Common Software Architectural Patterns in a nutshell

Ever wondered how large enterprise scale systems are designed? Before major softwa...

Sep 4, 2017 41K 137

Mario Bittencourt in Better Programming

"It's Not You, It's Me": Improving the Quality of Deliverables...

Automated testing is not enough and needs to be complemented by peer review.

Apr 4, 2023 331 2

See more recommendations