Rule of Software "Engineering"

How to not go crazy developing software in the real world.

Benjamin Shropshire
benjamin@precisionsoftware.us

April 27, 2021

Background

- These derive from the goals I want to share with my team.
- These are ideals. I don't expect projects to fully reach them.
- The real-world is built out of compromises. Trying to force a real world project into this will cause *other* issues.
- That said; having a target gives a direction to move towards.

Be careful what you measure, because that is what you will get.

Foreground

So what's the point? What is the overall motivation for this diatribe? What does this buy?

- **Sustainability** and **Maintainability** of a project.

But those are fuzzy term, *what do they mean?*

- Not just code (etc.) that doesn't break.
- Code that can evolve to address new requirements.
- Code that can react to late breaking "issues"[1].
- Code that can exploit newly acquired insight, experience and capabilities.
- Code that can escape its own legacy and choices.

[1] e.g. Meltdown, Spectre, Heartbleed

Introduction

- Rules for the build process.
- Rules for the code review process.
- Rules for testing.
- Rules for the release process.

But what do these terms mean?

There is something specific I'm referring to by these terms.
It's not the only thing people refer to with them, and those other things may be important and valuable, but those other things are not what I'm talking about here.
So I will be defining them below.

Rules for the build process:

Rule #0: Have a build process.

What is *not* a build process?

- A source code directory.
- A source code directory with a `README.txt` file.
- A source code directory with a shell script in it.

What is a build process?

- A stable, consistent, supported interface.

Why? What does this enabled?

- The developer only needs to think about a single "what" at a time. "What is the code doing?" or "What is the *build* doing?" Not both simultaneously.
- This stays true, even when starting to working on new and unfamiliar code.

Rules for the build process:

Rule #1: Use the same build process for "everything".

Okay, not *everything*, but wherever it makes sense.
Most any input-from-source/output-an-artifact step should be part of the build process. This can even include things like building the documentation and release notes.

Using the same build process uniformly:

- Allows developers to quickly jump into code they don't know.
- Allows portability/reuse.
- Reduce the number of things that need to be supported.

This makes the choice of build tool rather critical, as it needs to support a lot of use cases.[2]

[2] My personal, heavily biased, recommendation is Bazel.

Rules for the build process:

Rule #2: Use a scalable build process.

It needs both the *capability* and *capacity* to get the job done.

- Make sure it works.
- Make sure it does the things that are needed.
- Make it fast.
- Provision the capacity it needs.

Don't give the developers a reason to avoid using the provided build process.

"Build everything and run all the test to see what happens" should be an easy reflexive part of development.

Rules for the build process:

Rule #3: Use a hermetic build process.

Keep track of everything important to the build.

- The project source code.
- The dependent libraries (compiled or source code).
- The build process configuration(s).
- The tool chain configuration(s).
- The tool chain it self. (Themselves!)
- EVERYTHING!!![3]

[3] With apologies to Luc Besson.

Rules for the build process:

Rule #3: Use a hermetic build process.

Without this:

- Different developers will get different results.
- Just because a change works for one person doesn't imply it will even build for the next.

With this:

- Handing off a task to someone else can just be expected to work.
- Setting up a new developer's environment is little more than checking out the code.

Rules:

A quick aside ...

Rules:

Rule #0: Use source control.

Rules for the build process:

Rule #4: Use a reproducible build process.

- Re-building at any given point in history should get the "same" result as before.
- The only differences should be by explicit intent (metadata like build stamps etc.).

Without this:

- Figuring out what caused a change or a break becomes next to impossible.

With this:

- Going in and fixing something won't introduce uncontrolled changes.

Rules for the build process:

Rule #5: Commit to keeping the build clean.

Having this and keeping it working is not free.

- It needs buy-in from the people who use it.
- It needs buy-in from the people who set priorities.
- It needs people to own it, both the individual bits and the process as a whole.
- It needs to be a priority.
- Fixing a broken build needs to come before doing other things that are wanted, or even needed.

Rules for the build process:

Rule #0 Have a build process.

Rule #1 Use the same build process for “everything”.

Rule #2 Use a scalable build process.

Rule #3 Use a hermetic build process.

Rule #4 Use a reproducible build process.

Rule #5 Commit to keeping the build clean.

Rules for the code review process:

Rule #0: Have a code review process.

What is *not* a code review process?

- "Yo, I just submitted some code if anyone want to look at it."
- The design process.[4]
- Something to do just for particularly important/complex code.
- Rubber stamping.
- Hazing.

What is a code review process?

- An opportunity to get another perspective.
- A channel to distribute expertise.
- A way to maintain awareness.
- A demonstration of the importance placed on code quality.

[4] Design is important, but it's different than review.

Rules for the code review process:

Rule #1: Maintain a style guide.

A good plan violently executed right now
is far better than a perfect plan executed next week.

— George S. Patton

Having a style guide will help in a number of ways

- Avoid debates about trivia.
- Provide an authoritative source to refer to.
- Form a concrete bases from which to evolve best practices.
- Form a durable archive of experience and reasoning.

That said, a blind dogmatic adherence to a style guide can be worse than not having one at all.

Rules for the code review process:

Rule #2: Maintain defined owners for all code.

For any given part of the code base, there should always be (at least) three owners: one to review it, one to give a second opinion, and one who can be on vacation.[5]

- Any reasonable sized code base will be too large for any one person to know in detail.
- The owner is responsible for understanding that bit of code.
- Too many owners can result in "indecision by committee".
- Unowned code will end up being "someone else's problem".
- An owner is *not* the only person allowed to touch it.

[5] The same general pattern holds for other types of ownership as well.

Rules for the code review process:

Rule #3: Maintain automatic checks.

As far as possible automate the stuff covered by the review process.

- Auto-formatters.
- Lint.
- Presubmit checks.
- Static analysis.
- ...

This allows developers to clear these issue out of the way before sending things for review.

Note: The automated check must have very few false positives so that developers do not grow accustomed to ignoring everything.

Rules for the code review process:

Rule #4: Maintain those checks at head.

- Keep track of the automatic check results in the checked in state of the code.
- As the automatic checks evolve, plan to spend resources cleaning up old issues in existing code.

If it's worth reporting, it's worth putting effort into fixing. Putting effort into fixing something is an effective way to show people it's important.

Rules for the code review process:

Rule #5: Commit to doing code reviews.

Having clean code and keeping it working is not free.

- It needs buy-in from both sides of the process.
- Make reviewing code a priority.
- Make good code a priority.
- Plan for people to spend time on this.
- Make this one of the expectations of the job, not a tax.

Rules for the code review process:

Rule #0 Have a code review process.

Rule #1 Maintain a style guide.

Rule #2 Maintain defined owners for all code.

Rule #3 Maintain automatic checks.

Rule #4 Maintain those checks at head.

Rule #5 Commit to doing code reviews.

Rules for testing:

Rule #0: Have tests.

What is *not* testing?

- "Build it and try a few things."
- A list of things to try.
- A testing department.[6]

What is testing?

- Something that every developer can do on their own.
- Something with a formal definition of passing.
- Part of the build process (preferably).
- E.g. Unit and integration tests.

[6] Not that having one is a bad thing, that just something else.

Rules for testing:

Rule #0: Have tests.

Why? What does having good tests tests buy?

- It frees up focus from the extraneous, for the problem at hand.
- It makes it possible to work on code, without first needing to becoming an expert on it.
- It enabled cross cutting expertise. Dealing with a problem everywhere can cost $O(n)$ not $O(n^2)$.

Rules for testing:

Rule #1: Make it easy to find and run the tests.

Make finding them the same everywhere. Hunting for the test should be something nobody ever needs to do.

- If the tests are easy enough to find and run, running them will be regular part of the development process.
- Have definitive answer to the question of "what are the test?", or too much/too little time will be spent on testing.

Rules for testing:

Rule #2: Make the tests run quickly.

Shortening the edit/test/debug cycle has a super-linear effect on productivity.

- Quickly and definitively answering "did an edit fix the one thing or break another?" changes development processes.
- With fast enough tests, running them in a *known broken* state can be a reasonable way to answer questions about the code.

Rules for testing:

Rule #3: Make the tests run automatically during code review.

People should already run tests every time they send something for review or go to submit it. Just make it happen automatically.

- No need to remember.
- Provides a canonical answer for: "what are the right tests?"

Rules for testing:

Rule #4: Make the tests run automatically *after* review.

Trigger all the tests on a scheduled basis.

- Anything that is not monitored, will be broken.
- Running all tests before every submit becomes a bottle neck.
- Flaky tests will slip thought to head.
- Whenever a test fails, it's a simple matter to see the history.[7]

[7] This assumes that testing logs are being kept... You are, right?

Rules for testing:

Rule #5: Commit to keeping the test passing.

Having this and keeping it working is not free.

- It needs buy-in from the people who use it.
- Writing and running tests needs to be a regular, expected part of the development process.
- Sending off code with failing tests or without tests should be something that is actively avoided.[8]
- Broken tests being the norm will have people ignoring them.

[8] But don't go too far. People make mistakes; so plan for, and deal with that.

Rules for testing:

Rule #0 Have tests.

Rule #1 Make it easy to find and run the tests.

Rule #2 Make the tests run quickly.

Rule #3 Make the tests run automatically during code review.

Rule #4 Make the tests run automatically *after* review.

Rule #5 Commit to keeping the test passing.

Rules for the release process:

Rule #0: Have a release process.

What is *not* a release process?

- A document with a manual sequence of "build this", "run that", "update the thing", etc.
- Something that another team does once the project is finished.

What is a release process?

- Everything from checked in code through artifacts being ready to ship/deploy:
  - Build & test.
  - More tests (including stuff that isn't hermetic)
  - Deploy as canary.
  - Generate change-logs.
  - Push the docs.
  - Deploy to production.
  - etc.

Rules for the release process:

Rule #1: Ensure that the release process is easy to use.

- Eventually, it will have to be done by someone who is rushed for time, exhausted, sick or hungover.
- If it's hard to do when things are going correctly, then it will be impossible when it's most need.

Rules for the release process:

Rule #2: Ensure that the release process is effective.

- Don't give anyone an incentive to bypass it.
- Don't over engineer it.
- Keep it simple.
- Keep if flexible.
- Keep it transparent.
- *Things will go wrong.* People won't to be willing to start from square one when that happens, so allow manual flexibility.
- *Things will change.* People won't want to redesign things every time that happens, so allow for flexibility in the automation.

Rules for the release process:

Rule #3: Ensure that the release process is automatic.

- When things go right, the only action needed should be to kick off the process.
- Set up a cron job to kick it off regularly.
- Keeps it working. Keeps it fresh.
- When things break, they get noticed and fixed sooner.

Rules for the release process:

Rule #4: Ensure that the release process is safe.

- The process should automatically include enough tests that, if they pass, then things can ship.
- If any tests fail, the process should automatically block on fixing that.
- If the process is trusted, then taking exceptional action to override it can be treated as the exception, not the norm.

Rules for the release process:

Rule #5: Commit to keeping the release ready.

Having this and keeping it working is not free.

- This too needs buy-in.
- Someone needs to own the process and have the capacity to dedicate time to it.
- Having a release that is know to run consistently enables predictable deployments and responses to demands.
- Regular releases enable more quickly detecting issues further down the line.

Rules for the release process:

Rule #0 Have a release process.

Rule #1 Ensure that the release process is easy to use.

Rule #2 Ensure that the release process is effective.

Rule #3 Ensure that the release process is automatic.

Rule #4 Ensure that the release process is safe.

Rule #5 Commit to keeping the release ready.

Rules #0:

- Have a build process.
- Have a code review process.
- Have tests.
- Have a release process.

Rules #1: Good defaults:

- Use the same build process for “everything”.
- Maintain a style guide.
- Make it easy to find and run the tests.
- Ensure that the release process is easy to use.

Rules #2: Avoid blockers:

- Use a scalable build process.
- Maintain defined owners for all code.
- Make the tests run quickly.
- Ensure that the release process is effective.

Rules #3: Automate correctness:

- Use a hermetic build process.
- Maintain automatic checks.
- Make the tests run automatically during code review.
- Ensure that the release process is automatic.

Rules #4: Maintain the invariants:

- Use a reproducible build process.
- Maintain those checks at head.
- Make the tests run automatically *after* review.
- Ensure that the release process is safe.

Rules #4: Maintain the invariants:

- Use a reproducible build process.
- Maintain those checks at head.
- Make the tests run automatically *after* review.
- Ensure that the release process is safe.

Rule #4.1: Put the project status on a wall where everyone can see it.

Rules #5: Plan for the costs:

- Commit to keeping the build clean.
- Commit to doing code reviews.
- Commit to keeping the test passing.
- Commit to keeping the release ready.

The End

Extra

- Link: Impact on maintainability and refactoring for higher-level design features