Table of Contents

Introduction

Hi, I'm Ben Orenstein!

You might recognize me as the creator of Refactoring from Good to Great, How to Talk to Developers, or Refactoring Rails.

I've spent the last 10 years mildly obsessed with code and how to make it better.

I'm @r00k on Twitter.

About the challenge

Tiny wins, every day.

I've worked on a lot of codebases over the years, and I've discovered a recurring truth:

A steady drum-beat of small quality wins beats occasional large efforts every time.

Big-bang improvements are hard to schedule, difficult to justify, less fun, and don't work as well.

That's why I created this challenge.

Every work day (Mon-Fri), you'll receive a small exercise whose completion will improve the quality of the codebase you work on.

Sign up at: https://www.codequalitychallenge.com

Let’s start with a few meta recommendations/rules for the challenge:

1. The point of each exercise is to spend 20 minutes working on it, and to do this day after day. Much of the time, you will not be able to fully complete a challenge in the time allotted. This is okay. If you’re spending your 20 minutes each day, you’re doing it right. There’s no need for guilt, shame, or frustration here.
2. I strongly recommend you attempt each exercise as the first thing you do in your work day. If you plan to fit it in “at some point today,” it’ll be super easy to never get around to it. If possible, try to get the exercise done before you even check email or Slack.

About this document

This document is not part of the CQC. This is not how the Challenge works. In his last message at the CQC forum, dated 31st/Jan/2018, the author says he is not sure if he’ll run it again in the future. The aim of this document is to summarize the exercises and spread the knowledge.

#1 - Clean up your README

Spend 20 minutes improving your README

Why? Because it serves as your project's welcome mat. A good README will save you time and frustration when new people join your project (as internal team members, or open-source contributors). It also serves as the sales pitch for your app or library.

Here are some ideas of things to add to your README to flesh it out:

If your codebase is a library:

- The philosophy behind your tool
- Its place in the ecosystem
- A comparison to other similar libraries
- Lots of usage examples (it's hard to have too many of these)
- Animated gifs

If your codebase is an app:

- What role does it play?
- How is it deployed?
- Who on the team should people go to with questions?
- What credentials are needed?
- How do they install/boot/deploy/develop it?
- What's the development process?

For any project:

- FAQs
- Where should bugs be filed, and how?
- Who should be asked for PR reviews? (Are PRs welcome, generally?)
- How do you run tests?
- Is there CI?
- Screenshots

Here's a nice README template that you should steal from liberally:
https://gist.github.com/PurpleBooth/109311bb0361f32d87a2

And here's a collection of awesome READMEs that can help provide inspiration:
https://github.com/matiassingers/awesome-readme

#2 - Nuke some TODO comments

Today's exercise is short and sweet: grep your codebase for TODO comments.

When you find one, I recommend you do one of the following:

1. Is it out of date? Delete it.
2. Is it still relevant? Delete it and add it to JIRA.
3. Are you unsure? Do a little research and/or track down the comment's author and get an answer. Then do 1 or 2 :).

Why do this? Because code is a lousy place to track todos. When a todo lives in your code, it can't be prioritized or scheduled, and tends to get forgotten.

For extra points, submit a PR that deletes all the TODO comments and include links to the newly-created issue for each one. Should make for an easy review/approval.

Worth noting: spend 20 minutes on this task and then declare victory. Success is showing up and putting in the time, not necessarily deleting every comment. If you can only delete one comment in 20 minutes, you've still succeeded for the day (well done!)

#3 - Let's get rid of a warning

Get rid of a warning you've gotten used to ignoring.

Chances are, your code prints at least one warning during one of the following:

- Booting your code
- Running your tests
- Installing dependencies
- Deploying your app

These warnings are easy to learn to ignore with time, but there are two problems with that:

1. Many warnings (like a deprecation warning) will turn into full-blown issues later on. This will tend to happen at inconvenient times.
2. They act as a "broken window": an indication that your codebase isn't receiving the fastidious love and care it needs. These small messes quietly demonstrate to your team that messes are tolerated, and can often result in more. (More on "broken windows" https://blog.codinghorror.com/the-broken-window-theory)

So, today's task is to try to nuke at least one warning.

If you boot your app, run your tests, install your dependencies, and deploy with no warnings, congrats! Enjoy your day off.

Otherwise, see if you can eliminate a broken window.

#4 - Excise some unused code

Let’s track down some code that’s not actually in use (and delete it, naturally).

Those of you with statically-compiled languages will probably have little to do today, but in dynamic languages like Ruby and JavaScript, it’s pretty easy to end up with unused code laying around.

To track down your unused code, I recommend a tool like unused: https://unused.codes. I’ve personally used it to good effect in Ruby, and I believe it will work well with JavaScript and Elixir (see the docs).

#5 - Trim your branches

1. Run `git branch -r`. Marvel at all the old tracking branches that have been left in your local repo.
2. Run `git remote prune origin` to delete the local tracking branches that don’t exist on origin anymore. You might want to throw a `--dry-run` on there to confirm that git is going to do the right thing.
3. Re-run `git branch -r`. Better, right?
4. Now that your local repo is clean, take a look at the branches on origin by running `git ls-remote --heads origin`.
5. Delete any of your branches that are no longer needed with `git push origin --delete old_branch`
6. Maybe bug your coworkers to do 4 and 5, too.

#6 - Let’s extract a compound conditional

Extract a compound conditional.

This is best explained in a live-coding, video format. Fortunately, I have just the thing!

Please watch the first 5 minutes of this video: https://www.youtube.com/watch?v=0rsilnpU1DU&t=13s

The linked video is a sample of a course I created recently, and it contains my best explanation of how and why to extract compound conditionals.

Again, you only need to watch until 4:58.

For those of you that truly hate video, here’s a brief text version:

Compound conditionals look like this:

```
if foo && bar

  ...

end
```

or

```
if foo || bar

  ...

end
```

Your language might write these slightly differently, but the essential bit is that you have a conditional whose truth depends on the truth of two components combined with a boolean logic operator.

I’ve found that extracting these compound conditionals into a named method almost always improves the code.

In abstract terms, that looks like this:

Before

```
if foo && bar

  ...

end
```

After

```
if higher_level_concept?

  ...

end


def higher_level_concept?

  foo && bar

end
```

And here's a concrete example:

Before:

```
if user_created_account_today? && user_has_unconfirmed_email?

  prevent_user_from_posting

end
```

After:

```
if user_has_high_spam_risk?

  prevent_user_from_posting

end


private


def user_has_high_spam_risk?

  user_created_account_today? && user_has_unconfirmed_email?

end
```

I prefer the second version because it's more explicit. It takes an implicit idea: "users with new accounts and unconfirmed emails are more likely to post spam" and makes it explicit in the code. This is usually a big win, since it means there is more information in the code, and less that exists only in developers' heads.

Again, there's even more useful info in the video https://www.youtube.com/watch?v=0rsilnpU1DU&t=13s, and it's only 5 minutes, so I suggest watching it.

Today's challenge: find a compound conditional in your code (just grep for && and ||) and try extracting it into a well-named method. Try to make the new method explain a little more about what the compound conditional means at a higher level.

If the new name is a big enough improvement, consider committing/merging/opening a PR. If not, no worries. Just revert and remember this lesson next time you reach for && or ||.

#7 - Slim down a large class

Sometimes, despite our best intentions, a few classes in our system get large and unwieldy.

Today's exercise is to take a small step toward slimming down one of those classes.

First, use something like this to find your longest classes:

```
find ~/code/my_project -name "*.rb" | xargs wc -l | sort -rn | head
```

Then, pick one that looks like a good candidate and open it up.

Next, scan the file to look for opportunities to extract a new object.

When I do this, I'm looking for groups of methods that "clump together" in a related way.

Here are a few attributes that might identify "clumps" that may make sense to extract together:

1. Several methods that take the same parameter.
2. Several methods that access the same instance data.
3. Several methods that include the same word in their name.

When you see several methods that possess some of the above attributes, try extracting them into a new class and see if it feels like a worthwhile improvement.

An important caveat: this refactoring might be tough to pull off in 20 minutes.

Since you're working on a large class, you may find it has a lot of coupling that resists extraction.

Alternatively, you might not be able to find a good candidate for extraction.

In either case, here's a fallback task: improve SOMETHING about the class, even if it's tiny. Here are a few ideas:

- Delete a stray comment.
- Improve a name.
- Make something private if it's only called internally.
- Improve the formatting/style of any ugly bits (got any trailing whitespace or inconsistent newlines?).
- Slim down a long method.
- Delete some unused code.

Finally, please don't feel bad about any of the following:

1. How long your classes are.
2. The fact that you could only extract something small.

3. The fact that you couldn't find something good to extract.
4. The fact that you couldn't find a small thing to improve.

As always, this challenge is about showing up each day and taking a small step forward toward better code quality. Some days, you won't improve the code itself, but your mind. Attempting each exercise will prime you to perform better on your next project or task.

#8 – bin/setup

Remember those changes we made to your README on day 1? A common area for improvement was around setup instructions.

Know what's even better than a list of setup instructions to follow? A setup script that does them for you!

Here's an example that I've used on Rails apps to good success: https://gist.github.com/r00k/ab4dce37603cd94466c9955eee88ffe1

I name this script setup and throw it in the bin directory.

Ideally, this one command is all someone needs to run before developing on your app or using your tool. But even if you can't reach this ideal, a bin/setup script will save your users or team members time and frustration.

Today, please spend 20 minutes creating a similar script. You should be able to crib heavily from the example above, but here are a few ideas for things you might include:

- Download and install dependencies
- Create necessary databases
- Seed the database with useful data, like an admin account
- Set up useful git remotes
- Set configuration variables / copy config files
- Print helpful info

If you already have a setup script, try cloning your app into a new directory and running the script. Make sure it still works, and consider if you should add anything to it.

#9 - Run tests with wifi off

First, a warning: today's exercise is likely to generate tasks for you that will take longer than 20 minutes to fix. This doesn't mean something bad has happened, or that you've failed. Just file a ticket to remind you to complete the work later.

Today's exercise: run your test suite with your internet connection disabled.

Turns out, it's fairly easy to accidentally rely on an external service during test runs (I've done it many times). This won't just make your test suite slow, but brittle.

Chances are, you'll see a few cryptic failures when you try this.

In general, I recommend reaching for a tool like Webmock 5 to fix issues like these.

As a bonus, Webmock has a setting to disable external web requests during test runs. If you attempt to make a connection to the outside world, you'll get an exception. This will prevent you from writing internet-hitting tests in the future.

#10 - Investigate your slowest test(s)

Today's task is to investigate your slowest tests.

If you happen to use Ruby and RSpec, finding your 10 slowest tests is simply a matter of adding --profile to your test command. If you're using something else, hopefully a similar flag is only a quick google away.

Once you've identified your 10 slowest tests or so, give them each a once-over.

Some things to consider:

1. Are any of these tests duplicates? It's easier than you might think to accidentally write the same test more than once. You can't see that you've done this by reviewing a diff in a PR, so this type of duplication can often creep into your codebase over time.
2. Can any of these tests be replaced with a faster variant? If you're testing a conditional in the view, can you use a view spec instead of an integration spec? Can you stub expensive calls? Can you perform less setup? Can you hit less of the stack and still feel confident things work? (This video I recorded on integration vs. unit tests)[https://www.youtube.com/watch?v=kBOqaluDf2k] might be helpful.
3. Are all these tests still pulling their weight? Remember: slow tests act as a constant drag against your forward progress. Make sure the cost is outweighed by the benefits of keeping them around. I encourage you to be pragmatic. If a very slow test is verifying something that's not mission-critical, it's worth considering deleting it.

For Karma, try https://github.com/mlex/karma-spec-reporter

#11 - Improve a name

Improve one name today. Any name.

It can be a class name, a method name, a variable name, a constant name, a file name, anything.

If you just thought of a name you know needs improving, do that one.

If you can't find a name that could be improved, consider these questions:

- Do you ever refer to the same concept slightly differently in different spots?
- Have you noticed anywhere where a previous rename missed a few references?
- Pop open your schema. Are your database columns named consistently? (This is just a special case of the first one.)
- Is the name you'd use to describe a concept to a coworker the same as what's in the code?
- Is the name your customers would use the same as what's in the code?

#12 - Audit your dependencies

Crack open your Gemfile, package.json, setup.py, or whatever file your language/dependency manager uses.

Give it a slow scan. Ask yourself:

- Do you still need everything in there?
- Does anything need to be updated?
- Can you reduce a production dependency to a development/test one?
- Is your file nicely laid out and sorted alphabetically? Should it be?

Give it 20 minutes. Celebrate your continued focus on code quality.

#13 - Tidy your open Prs/issues

Please spend 20 minutes tidying up your PRs/issues/tickets/whatever.

If something is irrelevant, close or delete it.

If it something is old but might still be relevant, see if you can move it toward completion (or deletion). Try this: "This issue is quite old. I'm going to delete this early next week unless someone objects. That okay with you @whoever?"

If something is new and relevant, but not actionable, see if you can add or track down the information required to make it actionable.

I recommend working from oldest to newest.

As always, remember that the point isn't to clean up all the things, it's to spend 20 minutes chipping away at them.

#14 - Investigate long parameter lists

Do a search through your project for long parameter lists, where long is defined as more than three parameters.

Due to long parameter lists often being line-wrapped, you'll likely have to do some manual searching, but here's a naive regex that will find single-line method calls (in Ruby, at least) with at least four parameters:

(.*,.*,.*,.*)

You'll probably find that many of your long parameter lists appear when calling library or framework code.

However, you might find some in your own code. If you do, it doesn't necessarily mean there's anything wrong, but it's worth asking yourself a few questions about them:

1. Should any of the data that you're passing in be instance data instead? A clear indication this is true is if other methods on this object also require the same parameter.
2. Do you frequently pass several of these parameters together? If so, it's possible you have a Data Clump and could benefit from extracting a value object to contain them.
3. Are any of these parameters booleans? If so, you probably have a case of control coupling, and would do well to remove it.
4. Can any of these parameters be removed outright? You'd be surprised how easy it is to continue passing something into a method that no longer requires it.

By the way, the above questions are worth asking about just about any list of parameters, so consider them even if you tend to keep your parameter lists short.

#15 - Automate some repetitive action

Please spend 20 minutes automating/systematizing some action that you perform repetitively.

Here are a few ideas:

- Create a bin/deploy script to handle any tasks that you always perform before/after deploys. (One post-deploy task I like to include: open production in my browser to make sure I didn't just hose it somehow.)
- Create an alias for shell commands that you type frequently (git commands are good candidates). Here are my zsh aliases 8 for more inspiration.
- You know that thing you do all the time that makes you think "there has to be a better way"? Dig into that and see if you're right.
- Find a way to save yourself some keystrokes.

In general: find something you do at least a few times a week and see if you can make it more pleasant. Think of it as sanding down the rough edges in your development environment.

I think you'll find this one quite satisfying. When I automate an annoying task, I get a little burst of dopamine each time I use my new, improved method to accomplish it. One-time effort; continuous payoff.

#16 – Audit your schema

It's time to turn an eye toward your database schema.

Please spend 20 minutes reading through yours carefully.

Some things you might look for:

- Inconsistent column names.
- Missing indices for columns you frequently query by.
- Missing unique indices to ensure uniqueness.
- Missing null constraints.
- Missing foreign key constraints.
- Maybe even install bullet to detect N+1 queries in activerecord and mongoid.

Can you think of any other best practices that are missing from this list? Did you find an issue I missed? Please share it on the forum.

#17 - RTFM, please

Spend 20 minutes reading the docs for something you'd like to know a little better.

Ideally, this should be something that's already in your dev toolchain or used by your app. That'll let you apply what you learn right away.

You might want to investigate the docs for one of these:

- Your text editor. Vim users: type :h, search for "Editing Effectively" or "Tuning Vim", read one of the docs in that section.
- Your database. Any unused features you might leverage?.
- A library or framework you use frequently. Have recent releases added anything useful?
- Your shell. Can you optimize your workflows? Refactor a shell script? Improve your prompt?
- Your programming language. Anything new or unexplored there?

If you discover something new, try to use it right away. If you can't, maybe jot it down on a sheet of paper next to your keyboard so you remember to try it later. I do this often to teach myself new Vim commands. Once the new stuff is in my muscle memory, I throw the sheet away.