# Code Quality

## Static Analysis Tools

### Source Monitor

<http://www.campwoodsw.com/sourcemonitor.html>

## Naming

* Reveal the Intent:  
  The name of a variable, function or class should answer all the big questions: why it exist, what it does and how to use it.
* No Comments Needed:  
  If a name requires comment, it’s a bad name.
* Avoid Disinformation:  
  Call a group of students StudentGroup or Students not StudentList.  
  Calling a group a ‘List’ is a type of Hungarian notation if you’re actually using list type. Otherwise, it’s completely misleading!
* Make Meaningful Distinctions:  
  Don’t add noise words to names (CustomerData is the same as CustomerInfo or CustomerObject. Just Customer will do).
* Use Pronounceable names:  
  You should be able to talk about your code! No abbreviations! (no nm, prc or fnc)
* Use searchable names:  
  No magic numbers or one-letter names for variable (unless i,j in loops as counters)
* Avoid Encoding:  
  Encoding type or scope information into names simply adds an extra burden of deciphering.
* Class Names - Nouns:  
  Customer, Account, AddressParser etc. Avoid meaningless distinctions such as Manager, Processor, Data Info etc. Don’t use Verbs.
* Method Names – Verbs:  
  PostPayment, DeletePage, Save, isValid, getCustomer etc.
* Overloading:
  + Constructor Overloading:  
    When constructors are overloaded, use static factory methods with names that describe the arguments. For example,   
    Complex fulcrumPoint = Complex.FromRealNumber(23.0);   
    is generally better than   
    Complex fulcrumPoint = new Complex(23.0);
  + Don’t use overloading for different concepts. For example: don’t overload add to do something else other than the normal expected addition (like in 3+5).
* Don’t be cute and don’t Pun
* Pick one Word per Concept:  
  don’t have fetch, retrieve and get at different parts of the code. Pick one and stick with it.
* Use Solution Domain Names:  
  Whenever applicable, use computer science/mathematical terms where applicable e.g. AccountVisitor, JobQueue, BufferFactory etc.
* Use Problem Domain Names:  
  When there’s no appropriate solution domain name to use, use the problem domain.  
  Separating solution and problem domain concepts is part of the job of a good programmer and designer. The code that has more to do with problem domain concepts should have names drawn from the problem domain.
* Add Meaningful Context:  
  place names in context for your reader by enclosing them in well-named classes, functions, or namespaces. When all else fails, then prefixing the name may be necessary as a last resort (prefer to have address.firstName but if you have to, you can use addrFirstName).

## Code

* Set your line limit to 120 (clean code).

## Functions

“You know you are working on clean code when each routine turns out to be pretty much what you expected.”

* Small!
* Do one thing only:
  + If you can extract another function from with a name that is not just a restatement of its implementation, than your function does more than one thing.
  + If there are sections in the function (e.g. initiatlization, etc)
* Have no side effects! The function should do only what its name suggest and nothing more!
* One level of abstraction per function
* Use descriptive names. Can be long.
  + Encode the Arguments into the function name:   
    for example: AssertExpectedEqualsActual(expected, actual)
* Function Arguments: the fewer the better.
  + 0-2 are ok.
  + Try to avoid 3 or more. Consider passing in a structure instead of many related arguments.
  + Try to avoid output arguments. If your function must change the state of something, have it change the state of its owning object.
* Command Query Separation: Functions should either do something or answer something, but not both.
* Prefer Exceptions to returning error codes.
* Error handling is One thing! Extract the Try/Catch block into their own functions:

public void Delete(Page page)  
{  
 try  
 {  
 deletePageAndAllReferences(page);  
 }  
 catch(Exception)  
 {  
 logError(e);  
 }  
}

### Process

1. Write the test
2. Make it work
3. Refactor it. Split out functions, change names, eliminate duplications, shrink methods etc.

## Data Structures vs. Objects

* Objects hide their data behind abstractions and expose functions that operate on that data. Data structure expose their data and have no meaningful functions.
* As far as possible, it is better to define the data structure’s members as public member instead of defining them as private and setting setters/getters on them. This will make it clearer in the code using them that it’s using data structures and not objects.
* Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new classes without changing existing functions.
* Procedural code makes it hard to add new data structures because all the functions must change. OO code makes it hard to add new functions because all the classes must change.
* Mature programmers know that the idea that everything is an object is a myth. Sometimes you really do want simple data structures with procedures operating on them.  
  The trick is to know when to use data structures and when to use objects!

## Comments

“Don’t comment bad code – rewrite it.” Brian W. Kernighan and P.J. Plaugher

* The proper use of comments is to compensate for our **failure** to express ourselves in code.
* Good comments:
  + Explain the reason behind the code. E.g. in order to work-around an issue in the Flash chip, we have to…. Or See timing requirements in document….
  + Warning for the developers. E.g. don’t move this line of code because… or ‘Keep this function short since it’s running in ISR context’…

## Tests

When writing tests take into account: What will happen if the scenario is broken vs how much effort to maintain the tests?

## Tips

### Make the Implicit Explicit

Everything that is implicit in your code e.g.

Example:  
Unclear conditions like:

if created\_at <10 && user.is\_confirmed\_email?  
 display()

Can be moved into a private method:

private  
def good\_candidate\_for\_featuring?  
 created\_at <10 && user.is\_confirmed\_email?  
end

and in the main code:

if good\_candidate\_for\_featuring?  
 display()

### Policy Objects

If the conditions become too complex it might be a good idea to extract a policy object out of it:

private  
Class FeaturablePostPolicy  
 def initialize(post):  
 @post = post  
 end

def good\_candidate?  
 created\_at <10 && user.is\_confirmed\_email?  
 end

And in the main code:

def good\_candidate\_for\_featuring?  
 FeatureablePostPolicy.new(self).good\_candidate?

### Remove Comments

* TODOs – either turn it into a JIRA ticket or just delete it.
* Explanations – the code should explain itself.

The following comments are OK:

* Reference to the datasheet/requirement definition
* Comment that explain why we did something that looks weird. E.g. ‘this line has to be here because xxx’

### Prefer Exception over Silent Failing

This will prevent the application from assuming that everything is working fine and continue as normal when an error that needs handling occurred.

For the application to handle possible returned errors, it will need a lot of if-else that complicate the code.  
Using exception, you can separate the error handling from the ‘happy path’ and greatly simplify your code.

#### Defensive vs Optimistic Programming

* **Defensive Programming** – you should check and handle every possible error.
* **Optimistic Programming** – you should only handle the expected results and leave the application to throw exceptions if something unexpected happens.

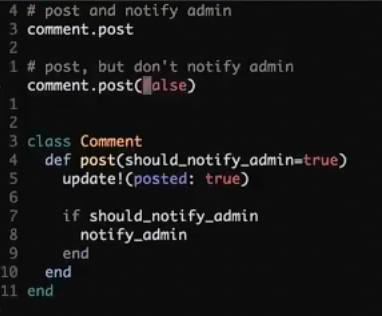
The truth lies somewhere in the middle:

* Defensive: You should check and handle all the expected error that can happen as part of the normal life of the application/API.   
  For example: if you are publishing a public API for handling movies, you need to check and handle the case where a movie doesn’t exist in the database and handle it correctly.  
  Note: it is always better to handle the error internally in a reasonable way instead of forcing the user to check and handle it.
* Optimistic: For unexpected errors that should not happen in the normal life of your application/API, you should throw an exception.  
  For example: if the API is a private one for your application only and you know that the user is choosing the movies from a drop down list, you don’t need to handle the case that the movie doesn’t exist in the DB and you can use the Single(…) function that will throw an exception if the movie does not exist.

### Don’t Use a Control Couple

\*\*\* I’m not sure that I agree with this section !!! This looks to me like over-complicating the code and creating more functions with (almost) the same functionality for no good reason!!!!

Control Couple is when you pass a Boolean argument into a method and control which of two paths it will execute:



Instead, you can define and use the following functions:

post  
post\_without\_notifying\_the\_admins

## Code Quality Challenge Cleanup

### Clean up 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

And here’s a collection of awesome READMEs that can help provide inspiration: [matiassingers/awesome-readme](https://github.com/matiassingers/awesome-readme) - A curated list of awesome READMEs

Set a timer for 20 minutes and get to it!

### Remove TODOs

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 whatever system you use to track work to be done, like GitHub Issues or Trello or Asana.
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!)

Finally, when you’re done, please share how old the oldest TODO comment you found.

### Get Rid of Warnings

Today’s exercise: 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/ 2](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.

### Trim your branches

 Run git branch -r. Marvel at all the old tracking branches that have been left in your local repo.

 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.

 Re-run git branch -r. Better, right?

 Now that your local repo is clean, take a look at the branches on origin by running git ls-remote --heads origin.

 Delete any of your branches that are no longer needed with git push origin --delete old\_branch.

 Maybe bug your coworkers to do 4 and 5, too.

### Extract 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 2](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 2](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 ||.

### Slim Down an Overgrown 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:

Several methods that take the same parameter.

Several methods that access the same instance data.

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:

How long your classes are.

The fact that you could only extract something small.

The fact that you couldn’t find something good to extract.

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.

### Create a bin/setup Script

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 7](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.

When you’re done, please chime in. Did you think of anything particularly creative to add to your script that others might want to steal? Please share!

### Run your tests with your 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 3](https://github.com/bblimke/webmock) 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.

Give it a try and let us know how it goes.

### Investigate your slowest test

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:

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.

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 2](https://www.youtube.com/watch?v=kBOqaluDf2k) might be helpful.

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.

### Improve One 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?

### 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?

Rubyists: maybe run [bundler-audit](https://github.com/rubysec/bundler-audit) to automatically check for gems with known vulnerabilites.

Is your file nicely laid out and sorted alphabetically? Should it be?

Give it 20 minutes. Celebrate your continued focus on code quality. Maybe post on the forum.

### Tidy your 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.

### 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:

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.

Do you frequently pass several of these parameters together? If so, it’s possible you have a [Data Clump 1](https://martinfowler.com/bliki/DataClump.html) and could benefit from extracting a value object to contain them.

Are any of these parameters booleans? If so, you probably have a case of [control coupling 1](https://robots.thoughtbot.com/types-of-coupling#control-coupling), and would do well to remove it.

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.

### Automate a 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](https://github.com/r00k/dotfiles/blob/master/zsh/aliases) 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.

### Audit your DB Scheme

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](https://github.com/flyerhzm/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.

### RTFM, Please

Today’s challenge: 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.

### Investigate high-churn files

(This challenge was created by guest contributor [*Giovanni Lodi*](https://twitter.com/mokagio). I recommend checking out his [*blog 2*](http://mokacoding.com).)

Today, please spend ~20 minutes looking into the churn your project has experienced.

In this case, churn refers the number of times a file has changed.

High-churn files are worth investigating. They’re not necessarily bad, but they might indicate a good refactoring opportunity.

Perhaps a file changes often because it is unclear and therefore buggy. Perhaps it’s doing too many things.

If you are using Git, you can measure how many commits there have been on each of your files using [this command 3](https://github.com/garybernhardt/dotfiles/blob/master/bin/git-churn):

git log --all -M -C --name-only --format='format:' "$@" \

| sort \

| grep -v '^$' \

| uniq -c \

| sort -n \

| awk 'BEGIN {print "count\tfile"} {print $1 "\t" $2}'

Consider saving it in an executable script. Call it git-churn, perhaps. This will make it easier to reuse and to pass parameters.

For example, what are the top 10 files in the core of your codebase that have changed the most in the past 3 months? Find out with:

git-churn --since='3 months ago' <core\_of\_the\_app> | tail -10

Today’s challenge is to spend investigate the files with the highest churn. Is there anything that stands out? Is it worth trying to make them more stable?

### Create/Update your snippets

(This challenge was again created by guest contributor [Giovanni Lodi](https://twitter.com/mokagio).)

Today’s challenge: spend 20 minutes getting snippets support configured in your editor. (If you’ve already done this, then spend some time adding a new one or two. Maybe delete some old ones you never use. Tend your snippet garden.)

I think it’s worth being fairly intolerant of unnecessary typing. It’s slow, error-prone, and puts mileage on your hands.

Let’s type less and get more done faster.

Here are some places to start. How to make snippets in…

[Atom 2](https://flight-manual.atom.io/using-atom/sections/snippets/)

[Sublime Text 1](http://docs.sublimetext.info/en/latest/extensibility/snippets.html)

[Xcode](https://nshipster.com/xcode-snippets/)

[Vim (via UtilSnip) 2](https://github.com/sirver/UltiSnips)

[VSCode 2](https://code.visualstudio.com/docs/editor/userdefinedsnippets)

If your development environment doesn’t support snippets, you can use a third party tool to bring snippets to all apps. On macOS, [Alfred](https://www.alfredapp.com/) and [Dash](https://kapeli.com/dash) are great.

<https://github.com/SirVer/ultisnips#screencasts>

### Start plugging a knowledge gap

Do you recall thinking something like “I really should know more about X”?

Today’s your chance to do something about it.

Please spend 20 minutes researching that thing.

Maybe it’s a technology in your stack that you feel a little guilty for not knowing.

Or maybe it’s a new tool that seems exciting but you haven’t had a chance to look into.

Don’t worry about a full exploration. Just chip away a little at your lack of knowledge. One step at a time

### Additional Ideas

* explore test coverage tools & spend 20 minutes backfilling some missing unit tests.
* pick an example refactor from the Refactoring book by Fowler & apply to your codebase
* write a blog post or explain to a coworker about something you recently learned
* spend 20 minutes learning some new tricks with your shell of choice (be it bash, zsh, fish, etc)
* share on the forum a command-line tool you use all the time that perhaps others might not know. Why is it useful? What do you use it for?
* look at the build times for your project, is there any way you can improve those build times? (caching, parallelizing build steps, etc)
* Check out [https://github.com/kamranahmedse/design-patterns-for-humans 1](https://github.com/kamranahmedse/design-patterns-for-humans) and think if any of those patterns could be applied to your project to help solve a maintenance problem you’re facing.
* <https://www.codependentcodr.com/category/posts.html> - additional coding challanges.