EFFECTIVE CODE REVIEW

Code review is a **key part** of the software engineering process. I'm going to be talking to you today about making code reviews as *effective* as possible.

JUSTIN SPAHR-SUMMERS

@JSPAHRSUMMERS

Let me introduce myself. My name is Justin Spahr-Summers, a.k.a., @jspahrsummers on basically every platform.

I've worked at smaller companies and larger ones, and been a key contributor to several successful open source projects in the Cocoa community, including Carthage, ReactiveCocoa, the Squirrel update framework, and the Mantle model framework for Objective-C.

In total, I've done thousands of code reviews—possibly *tens* of thousands—in both commercial and open source contexts.

IT'S NOT SOFTWARE ENGINEERING WITHOUT CODE REVIEW

Why this talk?

Code review is *fundamental* to our discipline. Without code review (and without other key fundamentals like testing, or CS theory), we're just *programming*—not engineering. We elevate our craft, and hold each other to a higher standard, through peer reviews.

POOR QUALITY **REVIEWS...**

- WASTE EVERYONE'S TIME TODAY
- WASTE EVERYONE'S TIME IN THE FUTURE
- PROVIDE A FALSE SENSE OF SECURITY

Many (I daresay *most*) teams are already doing some kind of code review. But if it's not *effective*, what's the point?

If your reviews aren't preventing technical debt, aren't improving the end result, and don't provoke good conversation... it's just burning everyone's time for little benefit—and your future self will pay the opportunity cost.

GREAT REVIEWS...

- MINIMIZE TECHNICAL DEBT
- IMPROVE THE ARCHITECTURE
- SHARE DOMAIN KNOWLEDGE
- PROVIDE TEACHING OPPORTUNITIES

Submitting a pull request should be the *start* of a conversation, not the end of one. Use code reviews to achieve all of these goals, and really dig deep into the essence of what you're trying to achieve. The result will benefit everyone!

I intentionally do not say that "great reviews prevent bugs." I think bugs should *primarily* be prevented through type systems, tests, etc., and not rely upon humans to catch before merging. But of course, some reviews will indeed catch some number of bugs.

FIRST PRINCIPLES

Hopefully we agree on how *important* it is to make reviews as effective as possible. Let's now discuss how to do that.

Before deciding on or describing any process, I like to identify the first principles. Process should follow principles, not the other way around.

For code review, the following are my principles.

GOOD CODE REVIEW
STARTS FROM THE SAME PERSPECTIVE AS
WRITING GOOD CODE

IF WE WERE TO WRITE THE BEST VERSION OF THIS, WHAT WOULD IT BE?

A great code reviewer is a great engineer, because being able to dig in and understand someone else's code, as well as how to *improve* it, is required all the time when both writing *and* reviewing code.

(Note that it doesn't work the other way—not all great engineers are automatically great reviewers! It's an additional skill, and requires commitment to become good at.)

UNBLOCK OTHERS

INCREASE YOUR TEAM'S PRODUCTIVITY

One of the most impactful things you can do is unblock your colleagues. PRs sitting open and unreviewed = lower productivity for everyone. I recommend establishing a maximum turnaround time for code reviews (e.g., 24 hours during the working week). If you're having trouble context switching, or bouncing back and forth between reviews and your own coding, establish a few recurring review times as a habit—like before starting your day, after lunch, and at the end of your day.

CRITIQUE THE CODE (NOT THE PERSON)

You may have heard the expression, "you are not your code." I prefer "*they* are not their code."

The code that someone submits should not reflect upon them as a person. Code reviews should never make it personal. It should be about the problem we're trying to solve, and whether this particular version of the change is the best way to do it.

DON'T ASSUME IT'S OBVIOUS

CODE CHANGES AND FEEDBACK BOTH NEED EXPLANATION

Many, many important things get omitted in communication because someone assumed, "this is obvious, I don't need to say it."

Always state your assumptions, and overcommunicate. Code authors, explain what you're doing and why! Code reviewers, explain the feedback you're giving, and why it will help!

THE PRINCIPLES

1. IMAGINE THE BEST VERSION OF THE CODE
2. UNBLOCK YOUR TEAM
3. CRITICIZE CODE. NOT PEOPLE
4. OVER-EXPLAIN EVERYTHING!

So just to recap, my first principles are to:
1. Imagine the best version of the code as the starting point for a review.
2. Unblock teammates and collaborators, to increase everyone's productivity.
3. Focus criticisms on code, and not make things personal.
4. And over-explain *everything*. This goes for authors *and* reviewers.

THE PROCESS

Okay. Let me share the process that I use now—the one I've honed over thousands of code reviews.

START AT THE HIGHEST LEVEL, THEN DIVE DEEPER...

1. INTENT
GOAL & SUMMARY

2. DESIGN
ARCHITECTURE & API

3. BEHAVIOR
TESTS & IMPLEMENTATION

I approach code reviewing in stages, "outside-in."

By starting at the highest level, I can short-circuit my review if I reveal anything that might require a major change to the pull request. There's no point in reviewing each line of code if the architecture is all wrong!

1. INTENT

Let's start with the *intention* behind the change, and make sure we agree on what we want this PR to actually accomplish.

WHAT IS THE GOAL?

IS THE EXPLANATION CLEAR?

Every single pull request summary should explain the goal of the PR. Otherwise, how will you as a reviewer evaluate it?

The author of a pull request will have the *most* context on the problem, so it's important that they can explain their goal clearly. And if the explanation doesn't make sense to you as a reviewer, **request changes**!

mplement `jest-haste-map` instead of `node-haste` #896

Closed **cpojer** wants to merge 10 commits into `facebook:master` from `cpojer:jest-haste-map`

Conversation 93 | Commits 10 | Checks 0 | Files changed 47

cpojer commented on Apr 15, 2016 • edited ▾ — Collaborator

This is a new haste map implementation for Jest which is much more scalable than node-haste.

node-haste2 isn't well designed and not scalable for short-lived services like Jest. The startup time of node-haste1 vs. node-haste2 on www is almost the same, both between 6 and 8 seconds which is not acceptable for our engineers. This implementation is attempting to accomplish a much reduced and more scalable startup time. It also has reduced scope – the goal of this is to only build a haste map and provide a way to resolve a one-level deep dependency tree, which is all that Jest really needs from node-haste. `jest-haste-map` can serve as an ideal basis for rewriting node-haste2 (into node-haste3!).

With this, the cold start time (building the entire haste map) is now about 14 seconds on www (that's acceptable) but the incremental invocation is only 2 seconds (**@kyldvs** will love me). I haven't heavily micro-optimized the JavaScript in `packages/jest-haste-map/src/index.js` and there is a bunch of data copying with `HasteResolver.js` – I believe I can get close to 1 second with these optimizations once I'm done. One of the goals I have is to allow tacking on data (list of mocks) to the haste map and serialize the haste map and read that one in directly in the workers instead of keeping two caches.

Here's an informative pull request summary from Christoph Nakazawa on Jest (a JavaScript testing framework). Although it could use a bit of structure (e.g., with headings), it's very clear in its goal...

facebook/jest#896

mplement `jest-haste-map` instead of `node-haste` #896

Closed **cpojer** wants to merge 10 commits into `facebook:master` from `cpojer:jest-haste-map`

Conversation 93 | Commits 10 | Checks 0 | Files changed 47

cpojer commented on Apr 15, 2016 • edited ▾ Collaborator

This is a new haste map implementation for Jest which is much more scalable than node-haste.

node-haste2 isn't well designed and not scalable for short-lived services like Jest. The startup time of node-haste1 vs. node-haste2 on www is almost the same, both between 6 and 8 seconds which is not acceptable for our engineers. This implementation is attempting to accomplish a much reduced and more scalable startup time. It also has reduced scope – the goal of this is to only build a haste map and provide a way to resolve a one-level deep dependency tree, which is all that Jest really needs from node-haste. `jest-haste-map` can serve as an ideal basis for rewriting node-haste2 (into node-haste3!).

With this, the cold start time (building the entire haste map) is now about 14 seconds on www (that's acceptable) but the incremental invocation is only 2 seconds (**@kyldvs** will love me). I haven't heavily micro-optimized the JavaScript in `packages/jest-haste-map/src/index.js` and there is a bunch of data copying with `HasteResolver.js` – I believe I can get close to 1 second with these optimizations once I'm done. One of the goals I have is to allow tacking on data (list of mocks) to the haste map and serialize the haste map and read that one in directly in the workers instead of keeping two caches.

The very first sentence gives you the goal right away—to introduce "a new haste map implementation ... which is much more scalable than node-haste." There's some assumed context here, but in this case, it refers to something all of the reviewers will be familiar with.

facebook/jest#896

mplement `jest-haste-map` instead of `node-haste` #896

Closed **cpojer** wants to merge 10 commits into `facebook:master` from `cpojer:jest-haste-map`

Conversation 93 | Commits 10 | Checks 0 | Files changed 47

cpojer commented on Apr 15, 2016 • edited ▾ Collaborator

This is a new haste map implementation for Jest which is much more scalable than node-haste.

node-haste2 isn't well designed and not scalable for short-lived services like Jest. The startup time of node-haste1 vs. node-haste2 on www is almost the same, both between 6 and 8 seconds which is not acceptable for our engineers. This implementation is attempting to accomplish a much reduced and more scalable startup time. It also has reduced scope – the goal of this is to only build a haste map and provide a way to resolve a one-level deep dependency tree, which is all that Jest really needs from node-haste. `jest-haste-map` can serve as an ideal basis for rewriting node-haste2 (into node-haste3!).

With this, the cold start time (building the entire haste map) is now about 14 seconds on www (that's acceptable) but the incremental invocation is only 2 seconds (**@kyldvs** will love me). I haven't heavily micro-optimized the JavaScript in `packages/jest-haste-map/src/index.js` and there is a bunch of data copying with `HasteResolver.js` – I believe I can get close to 1 second with these optimizations once I'm done. One of the goals I have is to allow tacking on data (list of mocks) to the haste map and serialize the haste map and read that one in directly in the workers instead of keeping two caches.

Then, the summary elaborates on why that goal is important. The existing implementation "isn't well designed and not scalable." "This implementation is attempting to [reduce] startup time."

Boom. Reviewers now have enough information to evaluate whether the goal is worth achieving (i.e., if this is the right problem to solve).

The next thing to figure out is...

facebook/jest#896

DOES IT SUCCEED?

HOW DO YOU KNOW?

Presuming you, as the reviewer, understand the goal of the PR, your next responsibility is to determine *whether this pull request actually achieves it*.

For example:

- Is the bug fix or new feature tested?
- How will you know if there's a regression?

These are a couple of the questions you (the reviewer) should have, and the author should have satisfying answers for you right there in the PR summary.

nitial version of jest-worker #4497

Merged **cpojer** merged 8 commits into `facebook:master` from `mjesun:jest-parallel` on Oct 4, 2017

Conversation 85 | Commits 8 | Checks 0 | Files changed 14

mjesun commented on Sep 17, 2017 • edited ▾ Contributor

This PR introduces a new module, `jest-worker`, intended to allow heavy task parallelization over multiple workers.

The module has a few advantages over the currently one used both in `jest` and `metro-bundler`:

- 100% `flow`-ified.
- 100% test coverage on it, all statements, methods and branches.
- Slightly faster than the currenly used one.
- Natively provides a `Promise` based interface, which allow us to avoid the extra wrapping layer in order to be used with `async`/`await.
- It only has one single dependency (`merge-stream`), which we could also remove.

Here's another pull request on Jest, from Miguel Jiménez Esún. All we need to know from this excerpt is that this is intended to be a performance improvement.

facebook/jest#4497

Performance test

It can be run by doing `node --expose-gc test.js` under `__performance_tests__`. Note that the percentage improvement shown (~ 10%) applies to 10,000 calls, meaning the performance improvement per single call is negligible. The test implements a `Promise` wrapper over the current implementation, so we can equivalently test both implementations as we use them in real scenarios.

```
------------------------------------------------------------------
jest-worker: { globalTime: 738, processingTime: 707 }
worker-farm: { globalTime: 885, processingTime: 866 }
------------------------------------------------------------------
jest-worker: { globalTime: 738, processingTime: 718 }
worker-farm: { globalTime: 865, processingTime: 849 }
------------------------------------------------------------------
jest-worker: { globalTime: 708, processingTime: 685 }
```

And we have *evidence* that it succeeds at improving performance, because the PR helpfully includes benchmarking results right in the description—along with the process by which anyone can run their own benchmarks.

facebook/jest#4497

Coverage

File ▲		Statements		Branches		Functions		Lines	
child.js		100%	27/27	100%	20/20	100%	4/4	100%	27/27
index.js		100%	67/67	100%	29/29	100%	11/11	100%	67/67
types.js		100%	5/5	100%	0/0	100%	0/0	100%	5/5
worker.js		100%	44/44	100%	13/13	100%	8/8	100%	44/44

This same PR description includes a picture of automated test coverage after the change. We have evidence that despite the performance improvement, no functionality has regressed. Great!

facebook/jest#4497

A GOOD SUMMARY SHOULD INCLUDE...

- BACKGROUND CONTEXT
- WHAT THE BUG OR FEATURE IS
- WHAT THE CHANGE ACHIEVES
- HOW THE CHANGE IS TESTED
- KNOWN LIMITATIONS OR ANYTHING STILL MISSING
- REQUEST CHANGES IF THE SUMMARY IS INCOMPLETE!

To flesh out the summary a bit more, it should include:
- Background context: any prerequisites necessary to understand the change
- Bug or feature: the goal of the pull request
- What it achieves: similar to, but not the same as, the goal
- How it's tested: include a description of automated tests, or at least a test plan!
- Known limitations: anything that is still to come, so the reviewer gets a picture of where this is heading next

Allow creating an alias for repositories #12000

Merged **sergiou87** merged 8 commits into `development` from `repository-alias` 25 days ago

Conversation 12 | Commits 8 | Checks 6 | Files changed 13

sergiou87 commented 26 days ago — Member

This is part of #7856

Description

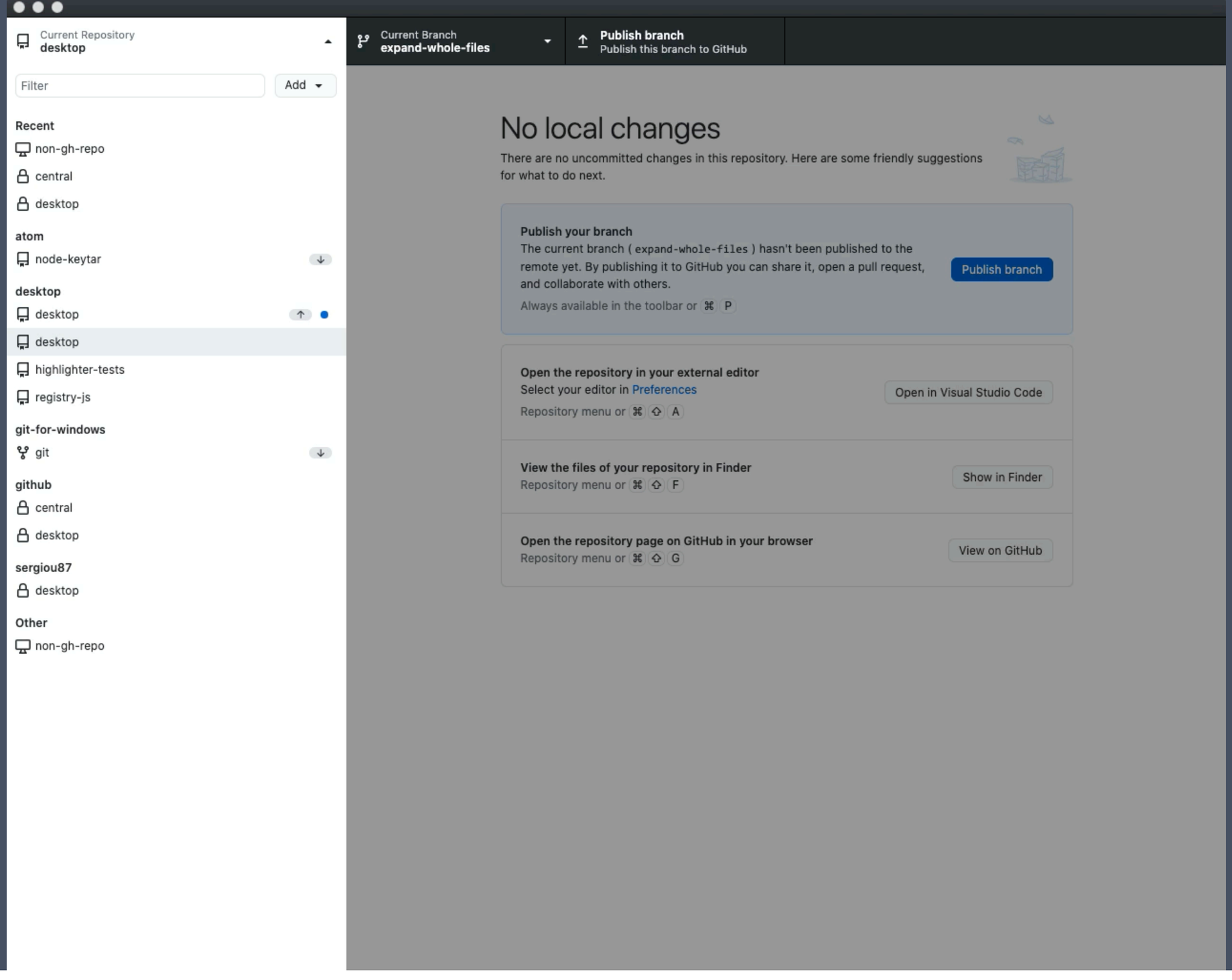
This PR adds initial support to create alias for repositories:

- New context menu actions to create, change or remove the alias of a repository from the repository list. Any character is valid.
- Filtering repositories now takes the alias into account.
- The original repository name is still visible by hovering over it, in the tooltip.

Screenshots

One more illustration, this time a pull request I randomly sampled from GitHub Desktop. I like this one because it's adding a new feature to an application, and... what's the best way to demonstrate that that's achieved?

desktop/desktop#12000



A video recording of that feature being used! The author has virtually let the reviewer try the feature themselves, without having to check out and build the code from scratch.

desktop/desktop#12000

ARE THE CHANGES & SUMMARY COMPLETE?

As the last step of understanding *intent*, evaluate (at a glance) whether the changes you see align with the explanation given in the summary. For example, are major architectural shifts explained?

Does this PR depend on something else? Does something else depend on this one? *Note that individual changes in a stack should still meet a high quality bar.*

2. DESIGN

That's our review of the intent. This is a good place to pause and reflect. If the intention of the pull request is not clear, or you disagree with the goal, there's no benefit to reviewing further.

But if you're on board so far, now it's time to review the high-level design of the changes.

ASK YOURSELF: HOW WOULD YOU DO IT?

Try to think through the bug or feature described, and mentally design your own solution for it.

If you didn't have this PR in front of you, how would you have done it? Does that highlight any gaps in the design you're reviewing?

REVIEW THE ARCHITECTURE

(THE COMPONENTS AND HOW THEY RELATE TO ONE ANOTHER)

- ARE THE ARCHITECTURAL CHOICES JUSTIFIED?
- DO YOU UNDERSTAND IT WELL ENOUGH TO USE OR EXTEND?
- WOULD EVERYONE ELSE BE HAPPY TO MAINTAIN THIS?
- DOES THE ARCHITECTURE MAKE YOUR LIFE EASIER?

If the author disappeared tomorrow, would the rest of the team be able to (and *want* to) work with this?

Will this be easy to run in production (whatever that means for your codebase)? Generally, simpler architectures are more maintainable.

REVIEW THE API

(THE CONTRACT FOR USING EACH COMPONENT)

- IS THE API UNDERSTANDABLE?
- DOES THE DOCUMENTATION TEACH THE READER HOW TO USE IT?
- IS THE API CONVENTIONAL?

Put yourself in the shoes of an API consumer. If you didn't have this PR in front of you, would you understand how to use it?

Remember the principle of "don't assume it's obvious:" what the author finds self-explanatory is often not the case for others!

On *convention*: if in Python, is it Pythonic? If inside a framework with its own conventions, does it match those? Although sometimes necessary, breaking convention introduces cognitive overhead.

IS THE DESIGN GOOD?

This is the most subjective part of the whole review, and partly just comes with experience, but here are some heuristics to help you answer this question.

Remember these for both quality coding *and* reviewing!

YAGNI

YOU AREN'T GONNA NEED IT

"Always implement things when you actually need them, never when you just foresee that you need them."—Ron Jeffries

"Do the simplest thing that could possibly work."

In other words, *don't over-engineer things*. Only make the design as complex as you need to for *today's* requirements, not the hypothetical future.

EASY

FAMILIAR OR APPROACHABLE

SIMPLE

FEWER CONCEPTS AND CONCERNS

When the two are in conflict, prioritize simplicity. If something is made *easy* just for the sake of familiarity, it can actually lead to more complexity, especially if the abstractions are leaky.

For example, platforms that try to make concurrency invisible to the developer can end up making things more complex (when bugs inevitably arise), even if it's *easier* to write the code at first.

See Rich Hickey's presentation, Simple Made Easy, as well as my favorite paper, Out of the Tar Pit.

PIT OF SUCCESS

MAKE THE RIGHT THINGS EASY & THE WRONG THINGS POSSIBLE

Does the API help consumers fall into the "pit of success," where doing the *right* thing is easy?

Are the default choices sensible and safe? Whenever there are multiple options, does the documentation clearly explain the consequences of each? Is the developer led toward the "right" choice?

It should be *hard* to mess things up! Strong static typing can help a lot here, by preventing logic errors and offering guidance to the user.

APIs like this can still offer escape hatches, to make "wrong" edge cases possible, but it's not a bad thing if they feel hard to use!

SOLID PRINCIPLES

- **SINGLE-RESPONSIBILITY PRINCIPLE**
  EACH THING SHOULD HAVE ONLY ONE RESPONSIBILITY
- **OPEN-CLOSED PRINCIPLE**
  BEHAVIOR SHOULD BE EXTENSIBLE WITHOUT MODIFYING CODE
- **LISKOV SUBSTITUTION PRINCIPLE**
  TYPES SHOULD BE REPLACEABLE WITH SUBTYPES
- **INTERFACE SEGREGATION PRINCIPLE**
  MANY SPECIFIC INTERFACES ARE BETTER THAN ONE ÜBER-INTERFACE
- **DEPENDENCY INVERSION PRINCIPLE**
  DEPEND UPON ABSTRACTIONS, NOT CONCRETE IMPLEMENTATIONS

Classic software engineering principles, still as relevant as ever.

3. BEHAVIOR

That's our review of the design. It's good to pause again here. If the design has significant flaws that need to be fixed, reviewing the tests and the implementation won't add any value, because they may completely change.

If the design mostly looks good, it's time to look at the implemented behavior.

REVIEW THE TESTS

- TESTS ARE ADDITIONAL DOCUMENTATION
- TESTS SHOULD PROTECT AGAINST REGRESSIONS
- TESTS SHOULD VALIDATE THE API CONTRACT
- TESTS NEED TO BE UNDERSTANDABLE
- ARE THERE MISSING TESTS?
- YOU CAN REQUEST CHANGES!

We'll start from the tests. This is sort of the code review analogue to test-driven development—verify that the tests make sense before checking the implementation.

Why tests first? They're a form of documentation, and they verify the API does what it says. This means we should look at what the *tests* are telling us, rather than the implementation.

If one of the tests fails, will you know how to investigate and resolve it? Sometimes a simpler test, covering less, is better than a complex test that we cannot hope to debug.

Don't wait until later for missing tests—they'll never get added!

REVIEW THE IMPLEMENTATION

- THIS IS THE LEAST IMPORTANT PART TO REVIEW!
  - BE SUSPICIOUS OF CONVOLUTED CODE
- WOULD YOU BE ABLE TO DEBUG THIS CODE?
  - DRY: DON'T REPEAT YOURSELF
    - DON'T REINVENT THE WHEEL
- DON'T IGNORE LINTERS AND WARNINGS

Finally, the implementation. But this is the least important part to review—correctness issues should mostly be caught through type systems and testing, not human reviewers. If the API contract is sensible and there are tests that validate it, any* implementation fulfilling the contract should be basically OK.

Does the code achieve its goal in the simplest, most maintainable way possible?

Code doing the same thing multiple times should factor that out.

Does something already exist to do (part of) what this implementation is trying to do? If you're a particularly helpful code reviewer, you can even offer suggestions for existing APIs, to help the author out!

If it ever feels like we're "swimming upstream," where there are flags and code smells left and right, maybe there's a good reason for it! Dig deeper—something that looks convoluted or hard might actually be the wrong thing to do.

** of course, there are exceptions; e.g., performance requirements, side effects*

THE PROCESS

1. INTENT
GOAL & SUMMARY

2. DESIGN
ARCHITECTURE & API

3. BEHAVIOR
TESTS & IMPLEMENTATION

So just to recap this process...

Keep this ordering in mind, so that you can go "outside in." You'll maximize your own efficiency at reviewing changes, and you'll be focusing your feedback on what is *most relevant* for the PR author.

OTHER PROTIPS™

Outside of the process itself, here are just a couple of things to add.

GIVE EFFECTIVE FEEDBACK

- ALWAYS REQUEST CHANGES OR ACCEPT
- BE PRAGMATIC FOR URGENT CHANGES
- SOLICIT SECOND OPINIONS
- PRIORITIZE YOUR FEEDBACK
- PROVIDE CONCRETE SUGGESTIONS
- EXPLAIN THE FEEDBACK

I've been saying "request changes" a lot. **That does not necessarily mean "you need to change this."** It's really just a signal that the author needs to take some action or provide more information (e.g., respond to a question) before the PR is ready for another review.

Likewise, **"accept" does not mean "this is 100% fine."** You can ask the author to make changes before landing, and we should trust each other enough that this will happen.

Be pragmatic: don't hold up critical fixes on a technicality.

On contentious topics, CC more people and ask them to weigh in! Make it a group decision, with consensus, instead of an argument to a stalemate.

Prioritize your feedback according to what you believe is most important to address ("definitely fix this" vs. "take it or leave it"), and provide suggestions for how the author can do that (e.g., "just a suggestion here").

Always remember to **justify your feedback**, just like you expect the PR itself to be justified. Point out how changing *this thing here* will lead to a better specific result, or end up improving *this other thing over here*.

TELL A STORY WITH YOUR COMMITS

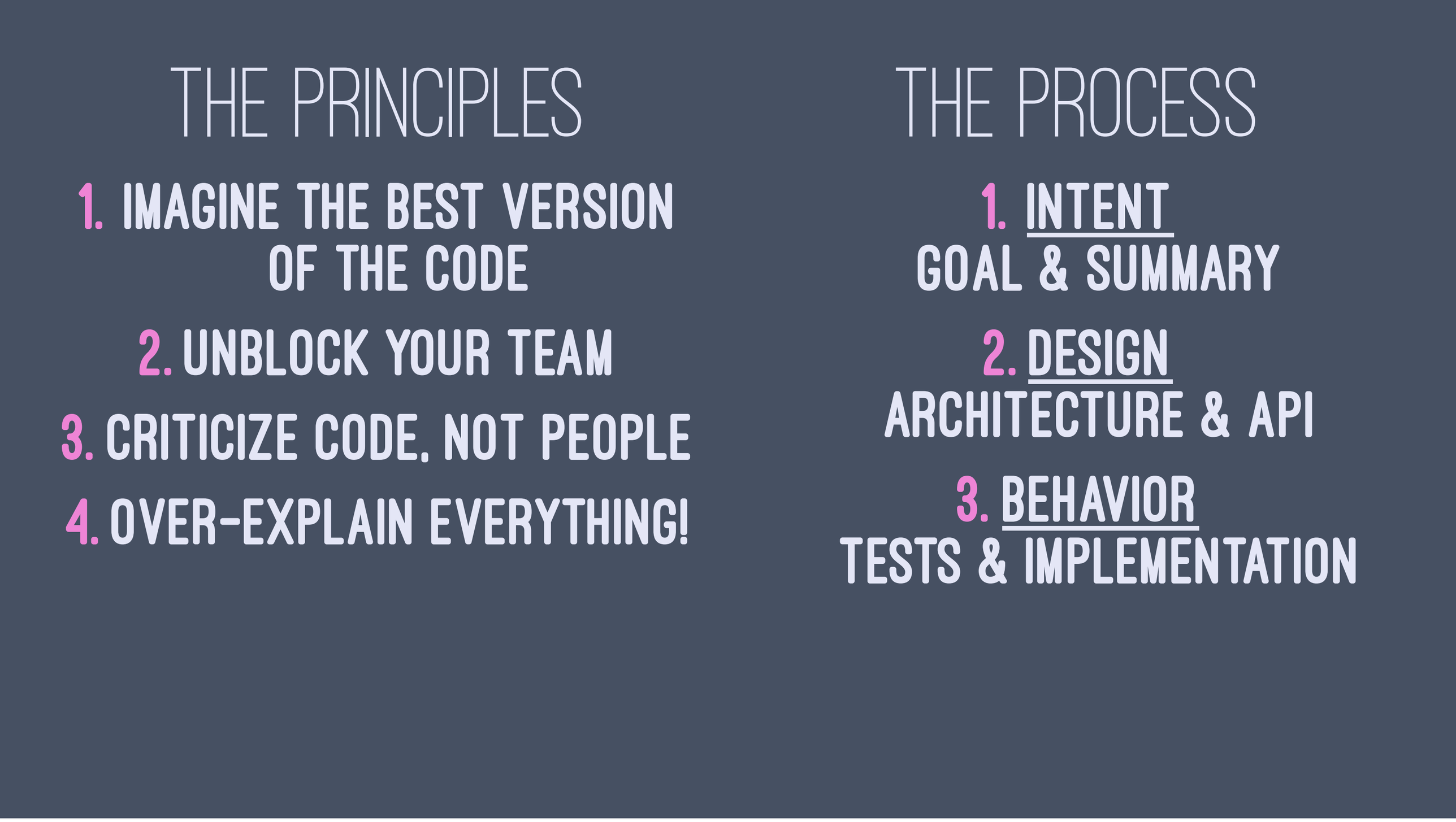
- EACH COMMIT SHOULD BUILD LOGICALLY UPON THE PREVIOUS
  - CLEAN UP AFTER YOURSELF:

```
git rebase -i
hg histedit
```

  - STACK DEPENDENT CHANGES

As a pull request *author*, you can apply all of the advice I've given so far as well! You can make your summaries clear and informative; you can evaluate and improve your design, tests, and implementation just like your reviewers will.

Here's one more thing that you, as an author, can do to make your reviewers' lives easier. By breaking down your changes into a logical sequence, they can see the individual parts of the change more clearly, and naturally follow the progression to a complete feature or bug fix.



So just to recap, my first principles are to:
1. Imagine the best version of the code as the starting point for a review.
2. Unblock teammates and collaborators, to increase everyone's productivity.
3. Focus criticisms on code, and not make things personal.
4. And over-explain *everything*. This goes for authors *and* reviewers.

And my process is to go *outside-in*, starting by reviewing the intent, then the design, and finishing by reviewing the behavior. I'll stop short at any point if I feel that major changes are necessary—this avoids wasting the reviewer's and the author's time.

QUESTIONS?

SLIDES AND NOTES AVAILABLE AT:
GITHUB.COM/JSPAHRSUMMERS/EFFECTIVE-CODE-REVIEW

THANKS TO
LIGHTRICKS AND BARAK YORESH
FOR INVITING ME TO SPEAK, AND TO
CHRISTOPH NAKAZAWA
FOR SENDING ME SOME GREAT EXAMPLE PRS!

Thank you all! I'm happy to take questions on anything and everything.