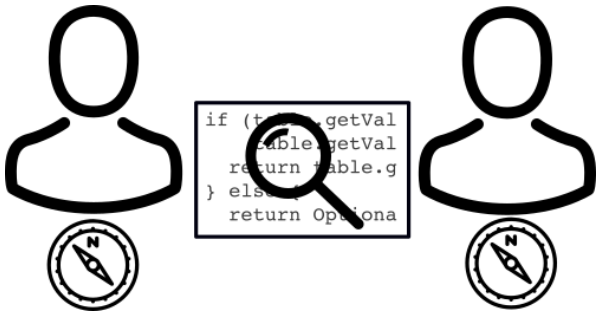
Philipp Hauer's Blog

Java Ecosystem, Kotlin, Engineering Management, Sociology of Software Development

Code Review Guidelines for Humans

POSTED ON JUL 31, 2018. UPDATED ON JUN 12, 2022

Code reviews are powerful means to improve the code quality, establish best practices and to spread knowledge. However, code reviews can come to nothing or harm interpersonal relations when they are done wrong. Hence, it's important to pay attention to the human aspects of code reviews. Code reviews require a certain mindset and phrasing techniques to be successful. This post provides both the author and the reviewer with a compass for navigating through a constructive, effective and respectful code review.



TL;DR

Code review guidelines for doing code reviews like a human.

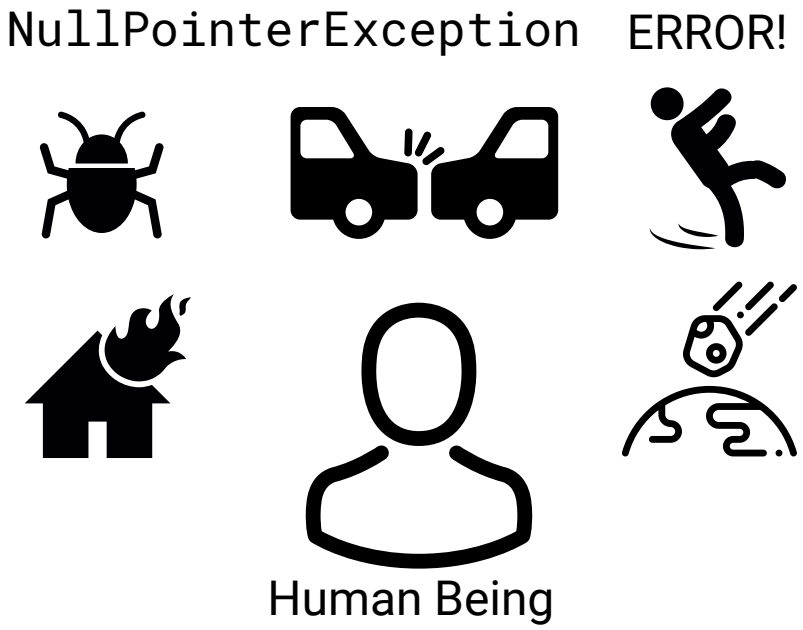
Workshop: Effective Code Reviews

I offer a workshop on how to establish an effective code review process and constructive code reviews in your team and company. Check out the details.

Code Reviews Guidelines For the Author

For you, as the author (or “developer”, “submitter”), it’s important to have an **open and humble mindset** about the feedback you will receive.

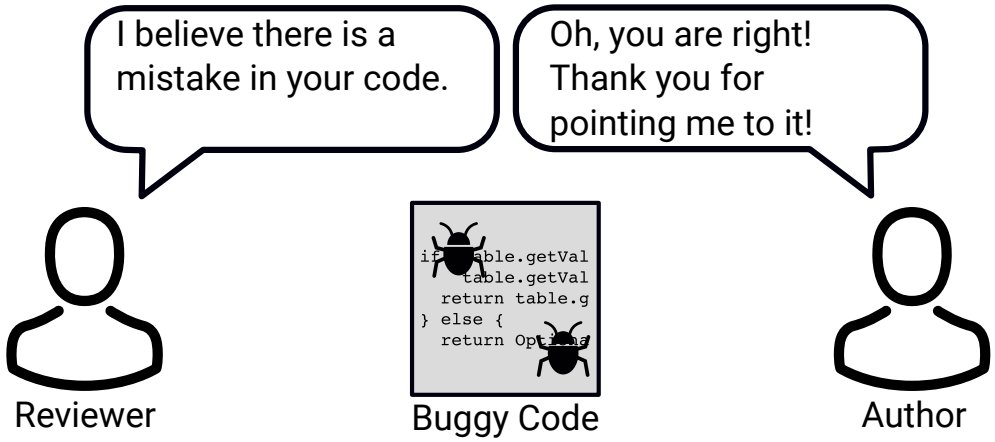
Be Humble



To err is human. Everybody makes mistakes.

We are humans after all. And humans make mistakes. That’s normal. So as long as software will be written by humans, it will contain mistakes.

This doesn’t mean that you should code carelessly or stop writing tests. But this mindset will take away the fear of mistakes and create an atmosphere where making mistakes is accepted and admitting them is desired. This, in turn, is important for criticism during a code review to be accepted. Otherwise, you may end up in endless justifications and rejections because mistakes may be seen as something forbidden and have to be kept hidden. This prevents the required openness for feedback.



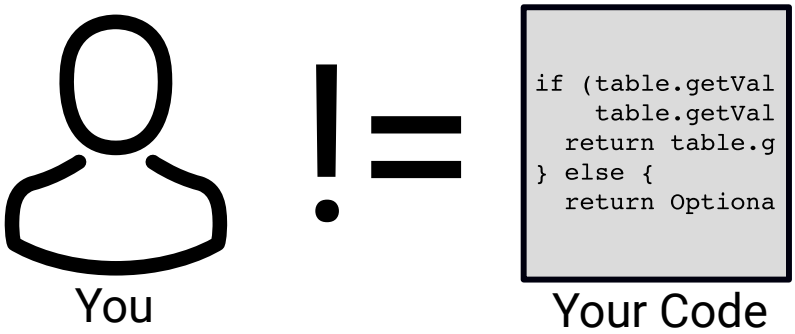
Making mistakes is accepted and admitting them is desired.

The key takeaway here is: **Be humble**. Mind that everybody's code can be improved. You are not perfect. So you have to accept that you will make mistakes. It doesn't matter how good you are, you can still learn and improve. Don't consider yourself as infallible and don't infer your professionalism and reliability as a software developer from infallibility and flawlessness. Admitting mistakes shows that you are professional, honest and after all a human being.

Moreover, I believe that the behavior of the team manager is crucial here. They are the role model for the whole team and should demonstrate a error culture by admitting mistakes in the public. And they should also call them "mistakes".

Besides, you should give the word "mistakes" a positive connotation by seeing them as opportunities to learn - and to call them so.

You Are Not Your Code



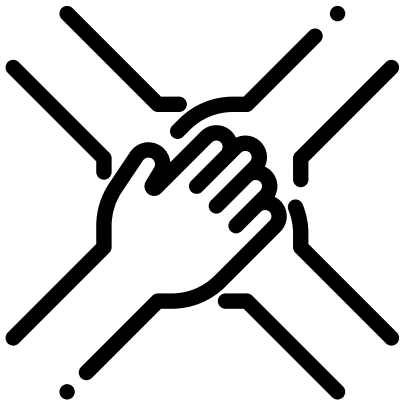
You are not your code.

Repeat after me: **You are not your code**. Criticism on your code is not a criticism on you as a human. Don't connect your self-worth with the code you write. You are still a valuable team member even if there are some flaws in your code.

In the end, programming is just a skill. It improves with training - and this improvement never stops.

You Are on the Same Side

> *"Criticism is almost never personal in a professional software engineering environment — it's usually just part of the process of making a better product"*. Fitzpatrick, Collins-Sussman: Debugging Teams, page 16



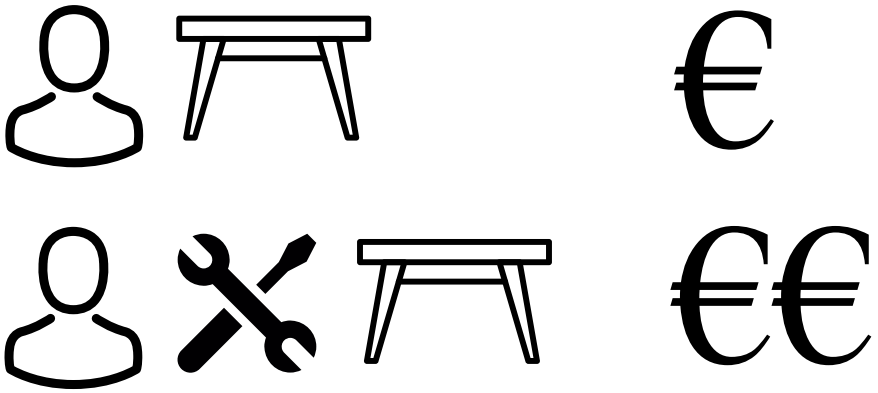
You are on the same side

When receiving feedback from the reviewer, always keep in mind that you and the reviewer are on the same side: You want to create a great product.

Mind the IKEA Effect

> The IKEA effect is a cognitive bias in which consumers place a disproportionately high value on products they partially created. Wikipedia

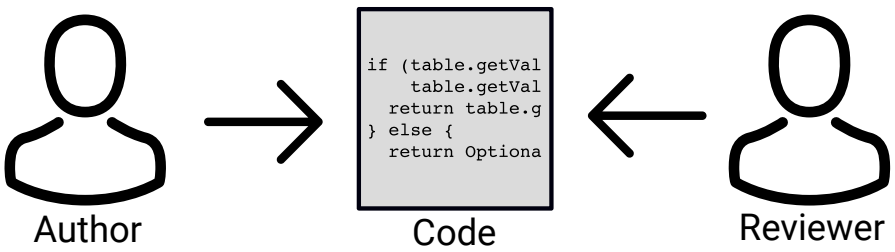
The effect was demonstrated by the following experiment: Two groups should price the value of IKEA furniture. One group got already assembled furniture, the other group had to assemble them first. The results showed that the second group was willing to pay 63 % more than the first group.



The IKEA effect let us place more value on things (furniture, code) we have created by ourself.

Applied to software development this means: We place more value into code that we have written. It might be harder for us to accept changes or removal of code that we have created. It's important to be aware of this bias when we receive feedback because we might be influenced by the IKEA effect.

New Perspectives On Your Code



A code review provides new perspectives on your code

Every developer has a different background, assumptions, knowledge, and experiences; and so does the reviewer of your code. It's totally natural that they see your code differently than you do. Moreover, they are not so familiar with the domain or concrete functionality that kept you busy the last days. That's great because this reveals code that was clear for you, but not for the reviewer.

```
// Reviewer: "When does this happen?"
if (article.state == State.INACTIVE) {
}
// Implicit knowledge here!
```

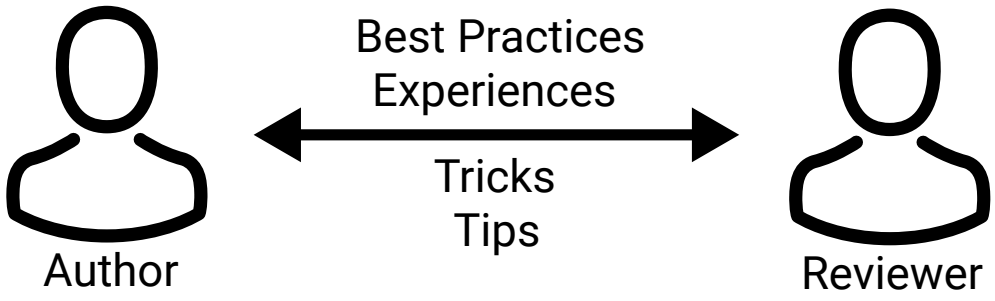
vs.

```
// Reviewer: "Ah, the state means out-of-stock"
boolean articleIsOutOfStock = article.state == State.INACTIVE;
if (articleIsOutOfStock) {
}
// make knowledge explicit by using expressive names
```

So code reviews **reveal the implicit knowledge** that is not expressed in the code yet because it appears natural for you. We are avoiding a tunnel vision.

But the best point here is: You don't have to be angry with yourself as you often simply ***can't* see those issues**. You only have one perspective. So just be thankful and embrace the opportunity to get a different perspective on your code. It's so valuable.

Exchange of Best Practices and Experiences



During a code review, the author and reviewer are exchanging best practices, experiences, tips, and tricks.

You and the reviewer are not only talking about your code - you are exchanging best practices and experiences. Code reviews are a great medium to establish and internalize good coding styles and best practices. And the exchange works in both directions. So consider code reviews as a valuable source of knowledge and an opportunity to learn.

Code Reviews Guidelines For the Reviewer

For you, as the reviewer, it's important to pay attention to the way you are **formulating your feedback**. The phrasing is extremely crucial for your feedback to be accepted.

Use I-Messages

Increase the acceptance of your feedback by using I-messages

> Wrong: “**You** are writing cryptic code."

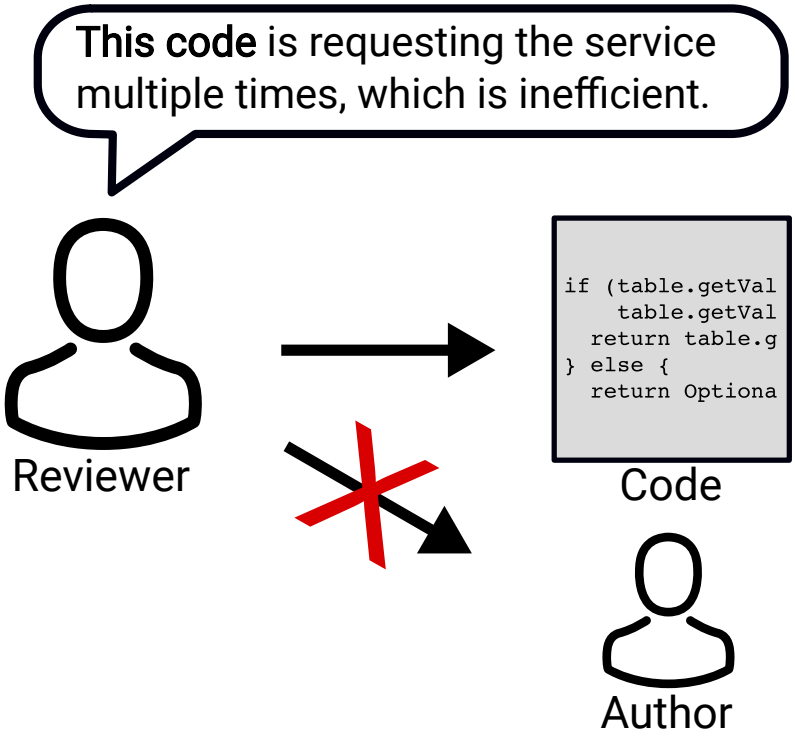
> Right: “It’s hard **for me** to grasp what’s going on in this code."

Always formulate your feedback from **your point of view** by expressing your **personal** thoughts, feelings, and impressions. Why? Because it’s hard for the author to argue against your personal feelings since they are subjective.

In contrast, You-messages sound like an insinuation and an absolute statement. It’s an attack on the author. They will definitely lead to justifications, rejections and a defensive stance. The author will not be thinking about how they can change, but rather how they can argue with you to show you that you are wrong. So the author will be less open for your feedback.

Using I-messages is the most important feedback rule in general and is clearly not limited to code reviews.

Talk About the Code, Not the Coder



Talking about the code increases the acceptance of your feedback, prevents pointless discussions and supports collective code ownership.

> Wrong: "**You're requesting** the service multiple times, which is inefficient."

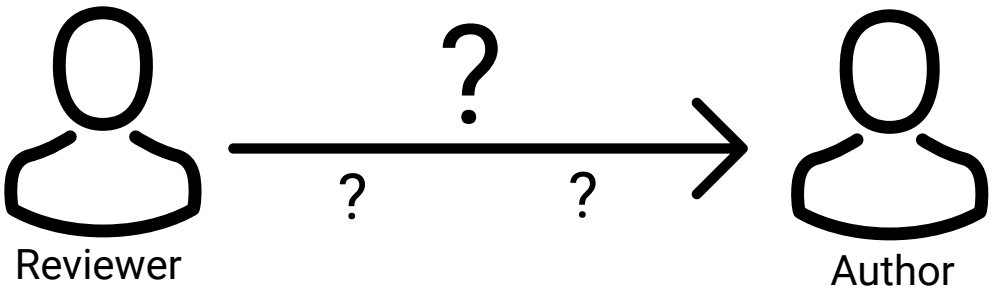
> Right: "**This code is requesting** the service multiple times, which is inefficient."

Take out the person in your feedback. Only talk about the code. Criticism on the code is much harder to take personally because you are simply talking about the code, an objective thing, and not the author. Again, this will improve the acceptance (as long as the author understands that they is not their code).

Moreover, this formulation prevents pointless discussions and finger-pointing like: "No, it wasn't me introducing this request logic. It was Dave who originally introduced this feature!".

And finally, talking about the code supports the notion of collective code ownership.

Ask Questions



Asking questions is a soft way to give feedback and to discover the author's intention

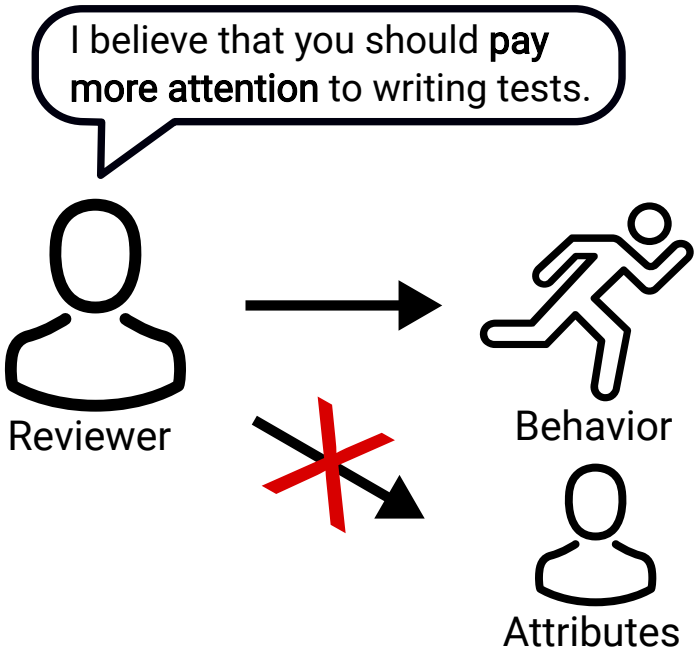
> Wrong: "This variable should have the name 'userId'."

> Right: "What do you think about the name 'userId' for this variable**?**"

Asking questions feels much less like a criticism; it's just a question that the author can answer. But it can trigger a though process which can end with accepting the reviewer's feedback. Or the author can come up with a new, even better solution. In both cases, the acceptance is much higher.

Moreover, by asking questions you can reveal the intention behind a certain design decision. Maybe there is a good reason for this which you haven't known. If so, you have discovered the intention without any judgement (due to incomplete knowledge) that may upset the author.

Refer to the Author's Behavior, Not Their Traits



Increase the acceptance of your feedback by only referring to the author's behavior.

> Wrong: "You **are sloppy** when it comes to writing tests."

> Right: "I believe that you should **pay more attention** to writing tests."

Another general feedback tip is to criticize only the behavior of the author, not their traits. Why? Traits stick to a human and are really hard to change. That's why that feedback often feels like an attack on the human being itself and will probably lead to resistance. The author will start to argue with you instead of thinking about how to improve the situation. Behavior, however, can be changed more easily. Criticism on the behavior is less likely to be perceived as a personal attack. The author will be more open to your feedback.

However, usually, it's **not required to talk about the author at all** (neither their traits or behavior) in a code review. I strongly suggest to use I-messages, talk about the code or ask questions.

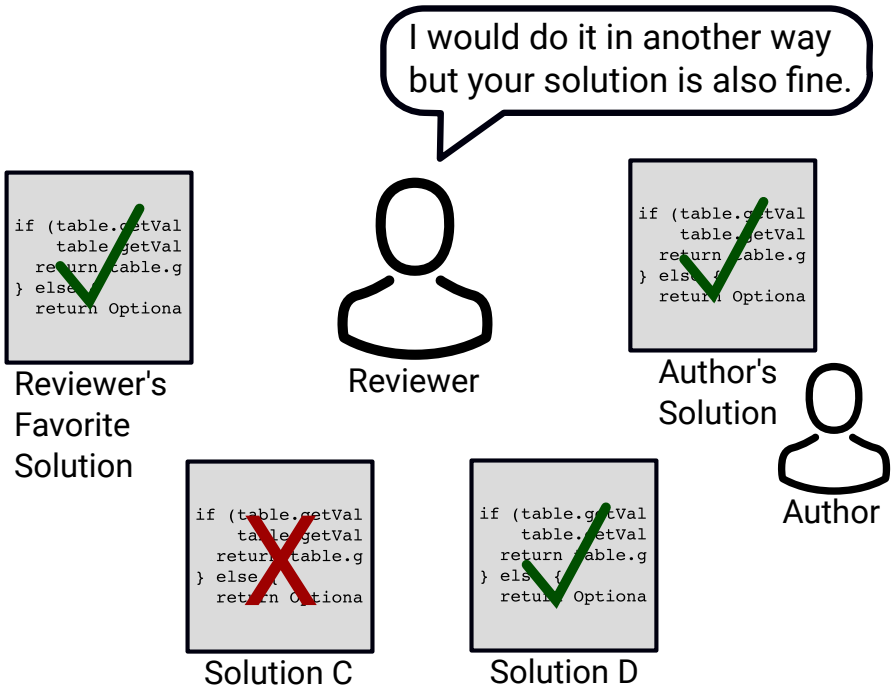
Mind the OIR-Rule of Giving Feedback

Another good tool for giving feedback is to structure the feedback into three parts: Observation, Impact and Request.

Example	Notes	
Observation	"This method has 100 lines."	Describe your observations in an objective and neutral way. Refer to the behavior if you have to talk about the author. Using an I-message is often useful here.
Impact	"This makes it hard for me to grasp the essential logic of this method."	Explain the impact that the observation has on you. Use I-messages.
Request	"I suggest extracting the low-level-details into subroutines and give them expressive names."	Use an I-message to express your wish or proposal.

The OIR-Rule is a great help for phrasing constructive and nonviolent feedback which doesn't trigger the author's defensive thread response.

Accept That There Are Different Solutions



Be aware of your personal taste, accept other solutions and make compromises

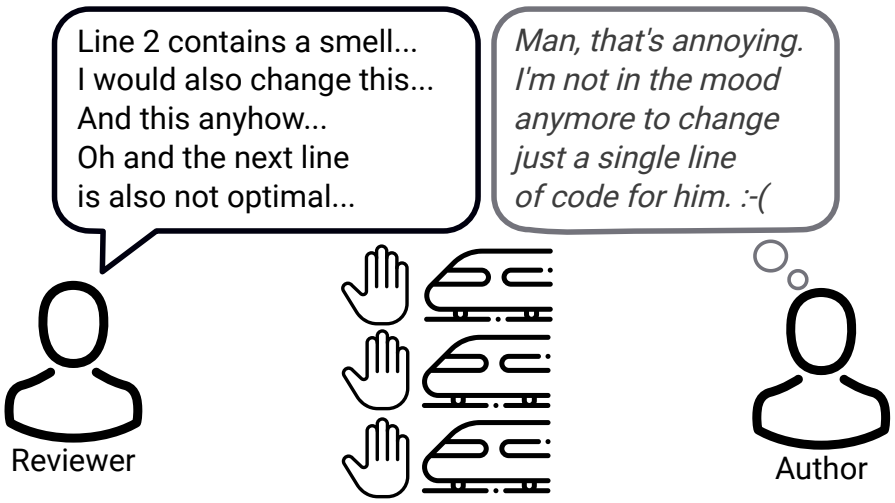
> Wrong: “I always use fixed timestamps in tests and you should too."

> Right: “I would always use fixed timestamps in tests for better reproducibility but in this simple test, using the current timestamp is also ok."

You have to keep in mind that there are always different solutions to a problem. Most likely, you’ll have a favorite solution, but the author’s solution may also be valid. Don’t force your solution on the author if their solution is also fine. **Distinguish between common best practices and your personal taste**. Mind that your skepticism may just reflect your personal taste and not an objectively wrong code. **Make compromises and be pragmatic**.

This mindset should prevent you from being uncompromising, pedantic and from annoying the author, which in turn reduces their openness to further feedback and may harm your relationship.

Don’t Jump in Front of Every Train



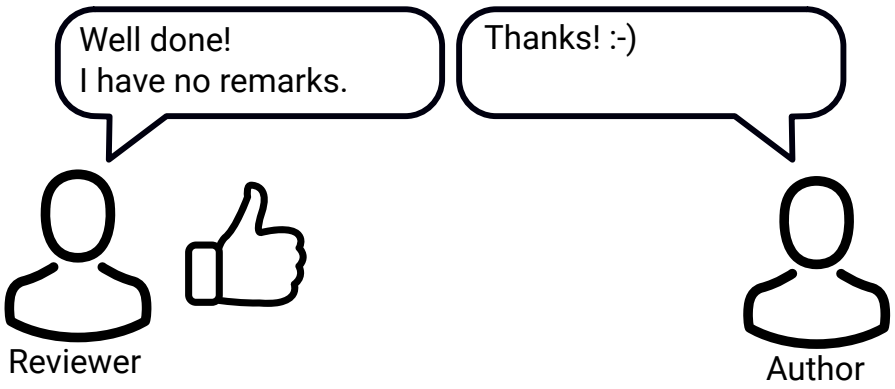
Don't jump in front of every train.

Don't be a pedant. Don't criticize every single line of code. Again, this would annoy the author and reduce their openness to further feedback and harm your relationship. And if your interpersonal relationship is destroyed, you have a much bigger problem than some not perfectly named variables. Instead, choose wisely the battles you are going to fight. Focus on the flaws and code smells that are most important to you.

I really like the metaphor used in Debugging Teams to emphasis this hint:

> *"Every time a decision is made, it's like a train coming through town — when you jump in front of the train to stop it you slow the train down and potentially annoy the engineer driving the train. A new train comes by every 15 minutes, and if you jump in front of every train, not only do you spend a lot of your time stopping trains, but eventually one of the engineers driving the train is going to get mad enough to run right over you. So, while it's OK to jump in front of some trains, pick and choose the ones you want to stop to make sure you're only stopping the trains that really matter."* Fitzpatrick, Collins-Sussman: Debugging Teams, page 72

Praise



Don't forget to praise.

Don't forget to express your appreciation if you have reviewed good code. Praising doesn't hurt you but will motivate the author and improve your relationship.

But your praise should be specific, concrete and separated from your criticism. Use different sentences and avoid sandwiching:

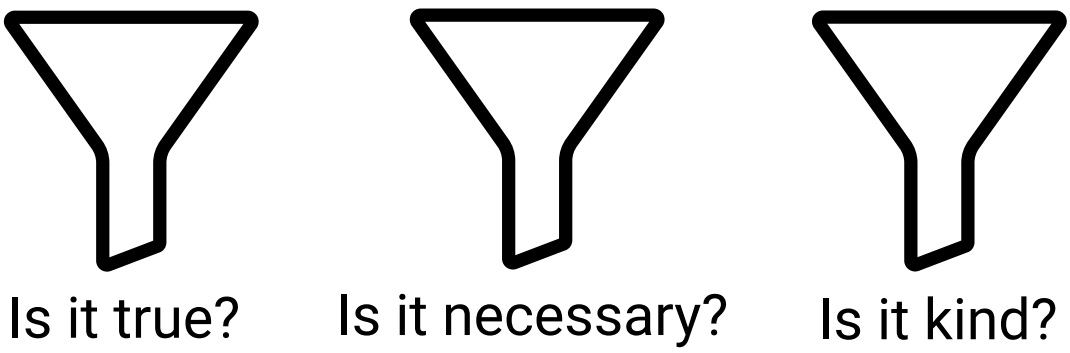
> Wrong: "Most of your code looks good, but the method `calc()` is too big."

> Right: "I really like the class `ProductController` , Tim. It has a clear single responsibility, is coherent, and contains nicely named methods. Good Job!
> Despite this, I spotted the method `calc()` which is too big for me."

And last but not least: It's totally fine to say: "Everything is good!". *No code changes* is a valid outcome of a code review. Don't feel forced to find something in the code.

Three Filters For Feedback

April Wensel proposed a great approach in her talk 'Compassionate (Yet Candid) Code Reviews' to check your feedback. Before giving feedback, ask yourself:



Always ask yourself, if your feedback is true, necessary and kind (Contribution: April Wensel).

Those questions aim at many points that have already been covered in the previous sections. However, I like them because they provide a great mental crash barrier during code reviews.

Is it True?

> Wrong: “You should use getter and setter. This code is wrong."

Is it true? This statement assumes an absolute truth, which rarely exists. Avoid the words “right”, “wrong” and “should”. Often, you only refer to your opinion. If so, say it:

> Right: “In this case, I would recommend using getter and setter, because…"

Again, you can ask questions:

> Right: “Did you consider to use getter and setter?"

If it’s a fact, refer to a source (an official or the team’s style guide):

> Right: “According to the Java style guide…"

Is it Necessary?

> Wrong: “There is a space missing here."

Is it necessary? I believe that there are more important things to talk about in a code review than missing spaces. Nagging tends to annoy the author. This corresponds with the advice to don’t jump in front of every train.

> Wrong: “**This code sends a chill down my spine**, but I see your intention."

Is it necessary? The first part of the sentence has no sense. The reviewer only tries to show how cool they are. The only effect is that the author will feel bad and attacked. So always check your intention: Are you trying to help or boosting your ego?

> Wrong: “We should refactor this whole package."

Is it necessary to refactor the whole package in the scope of the current feature and code review? It's fine to detect those *big* refactoring needs, but you should work on them separately. Consider to open a ticket or to have a dedicated meeting or chat with the whole team. Choose an appropriate channel.

> Right: "Let's discuss the refactoring in a dedicated meeting."

So the necessary-question has several aspects: To avoid nagging, unnecessary comments, and out-of-scope work.

Is it Kind?

> Wrong: "A factory is **badly over-engineered** here. The **trivial** solution is to **just** use the constructor."

Is it necessary? No, and is it kind? Absolutely not! This statement is shaming. It gives the author the feeling of being stupid.

Being kind does *not* mean that we grab us by the hands, sing "Kumbaya My Lord", and stop saying something unpleasant. The point is, that being kind is a smart strategy to give feedback that will be accepted. It's efficient because you don't trigger someone's defensive reaction.

The following statement has the same message but without any shaming:

> Right: "This factory feels complicated to me. Have you considered to use a constructor instead?"

The Code Review Cheat Sheet

Rules For the Author

For the author, it's important to have an **open and humble mindset** about the feedback they will receive.

- Be humble!
  - Mind that everybody's code can be improved.
  - You are not perfect.

    - Accept that you will make mistakes.
    - No matter how you good you are, you can still learn and improve.
    - Don't infer your professionalism and reliability as a software developer from infallibility and flawlessness.
- You are not your code
    - Programming is just a skill. It improves with training – and this never stops.
    - Don't connect your self-worth with the code you are writing.
- Mind that finally, the reviewer wants the same as you: Creating high-quality software. You are on the same side.
- Mind the IKEA effect. You might place a too high value on your own code.
- Consider feedback as a valuable new perspective on your code
    - It reveals your implicit knowledge that is not expressed in the code yet because it appears natural for you.
    - It avoids a tunnel vision.
- Code reviews are a valuable source of best practices and experiences
- Code reviews are a discussion, not a dictation. It's fine to disagree, but you have to elaborate your reservations politely and be willing to make compromises.

Rules For the Reviewer

For the reviewer, it's important to pay attention to the way they **formulate the feedback**. This is extremely crucial for your feedback to be accepted.

- Use I-messages:
    - Right: "It's hard for me to grasp what's going on in this code."
    - Wrong: "You are writing cryptic code."
- Talk about the code, not the coder.
    - Right: "This code is requesting the service multiple times, which is inefficient."
    - Wrong: "You're requesting the service multiple times, which is inefficient."
- Ask questions instead of making statements.
    - Right: "What do you think about the name 'userId' for this variable**?**"
    - Wrong: "This variable should have the name 'userId'."
- Always refer to the behavior, not the traits of the author.
    - Right: "I believe that you should pay more attention to writing tests."
    - Wrong: "You are sloppy when it comes to writing tests."

- Accept that there are different solutions
  - Right: “I would do it in another way, but your solution is also fine.”
  - Wrong: “I always do it this way and you should too.”
  - Distinguish between common best practices and your personal taste.
  - Mind that your criticism may just reflect your personal taste and not an objectively wrong code.
  - Make compromises and be pragmatic.
- Don’t jump in front of every train
  - Don’t be a pedant. Don’t criticize every single line in the code. This will annoy the author and reduce their openness to further feedback.
  - Focus on the flaws and code smells that are most important to you.
- Respect and trust the author
  - Nobody writes bad code on purpose.
  - The author wrote the code to the best of their knowledge and belief.
- Mind the OIR-Rule of giving feedback
  - Observation - “This method has 100 lines.”
  - Impact - “This makes it hard for me to grasp the essential logic of this method.”
  - Request - “I suggest extracting the low-level-details into subroutines and give them expressive names.”
- Before giving feedback, ask yourself:
  - Is it true? (opinion != truth)
  - Is it necessary? (avoid nagging, unnecessary comments and out-of-scope work)
  - Is it kind? (no shaming)
- Be humble! You are not perfect and you can also improve.
- It’s fine to say: Everything is good!
- Don’t forget to praise.

Workshop: Effective Code Reviews

I offer a workshop on how to establish an effective code review process and constructive code reviews in your team and company. Check out the details.

Further Reading

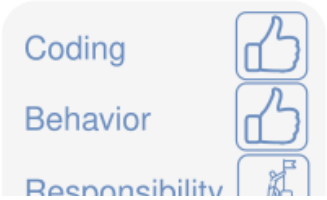
- I highly recommend the book Debugging Teams by Brian Fitzpatrick and Ben Collins-Sussman. Without exaggeration, this book inspired me.
- April Wensel: Talk 'Compassionate (Yet Candid) Code Reviews'
- Matt Stine: The Ten Timeless Commandments of Egoless Programming

---

You can follow me on Twitter or Mastodon, subscribe my newsletter or request my consulting or training.

Related Posts

Convincing Your Management to Introduce Kotlin



Better Performance Reviews for Developers with a Skill Matrix

Effective Staff Appraisals with Employee Journey Maps

How to Motivate a Team of Software Developers

This entry was posted in Leadership, and tagged with Code Review, Clean Code, Human, Feedback,

---

Comments

Write a Comment...

Michael Payne 31 days ago

I thought this was a really good article. I would like to see a follow-up article assuming knowledge of the information in this one (with maybe a summary or similar and a link) , but explaining any changes in this dynamic for remote teams, when face to face communication is not possible. When / not / how to use meeting software (Zoom etc.) Kind of a "post-Pandemic" update.

Reply

Alina Smith 11 months ago

I really enjoyed this blog. It is very informative for engineers. But for becoming an engineer you have to study hard and complete all the tough assignments.
If you want engineering homework help online you can visit https://tutorchamps.com
Where you can find engineering assignment experts who can help you with engineering assignments.

Reply

Xander 17 months ago

Code reviews can be traumatic emotionally at best. Esp. if changes requested are trivial and would not provide any additional value. As mentioned above resulting in out of scope work. Not all changes need coded unit tests like text changes. Would say that code that could effect security, cause a potential data loss or data integrity issue or even architecturally altering the pattern that existed before are examples that need unit tests. Above all Phillip Hauer has done a very good job in showing both reviewer and the reviewed that this process can engender +ve team bonding rather that being confrontational OR a stressful divisive -ve force. Happy coders make for Better products.

Reply

Eugenio Romano 3 years ago

HI @disqus_liMcuzsmUM:disqus do you mind if i use it in the wiki of https://github.com/Alfresco... as policy for the code review?

Reply

Aleksi Sjöberg 4 years ago

Trying to start implementing more code review practices within our team. This post is a good resource for tips. Props for informative pictures and TL;DR summary :)

Reply

Philipp Hauer 4 years ago

Thank you very much! :-) I'm glad you like my post. And good luck for your code reviews.

Reply

Michael Parker 4 years ago

A few bits I would add:
1. I actually prefer not to include my own feelings when reviewing cryptic code. As a reviewer my objective is not to make my life easier, but to act in the best interests of the team, and make it clear thats what you're doing. I often use the example of people new to the team, or helping junior developers. I prefer comments like 'I think we can improve the readability by extracting a variable here called xyz' or 'Lets extract a function to reduce nesting here'. If you need people to buy into the culture of readable code, then that needs to happen outside of PRs, so you shouldnt need
2. People don't write unreadable code deliberately, they generally need mentoring (if it was easy they wouldnt mind doing it so much). So offer to help! If a large chunk is unreadable, how about 'I'm concerned that the team would struggle to understand this section, it appears quite complex. I wonder if we could pair on spiking out some alternative solutions?' or how about 'I think there might be a simpler solution to this - would you be open to working this through with me?'
3. Don't try to solve big problems or conflicts in pull requests / online reviews. If someone has rewritten half your codebase, don't add 100 comments to every last line. 'Before reviewing in depth, it'd be good to understand the background to this PR, can we catch up face to face?'
4. If a comment thread (argument / debate) goes beyond 3-4 comments, resolve it face to face. Often I don't even reply to the thread, I just turn up to someone's desk "Hey it'd be great to get your pull request in - do you have 5 mins to run through a few of the discussion points?'
5. Often changes you don't agree with are the result of problems elsewhere, such as lack of communication / interaction, differing code styles across teams, lack of agreement on standardisation, lack of pair programming / mentoring, etc. For every review comment it's worth thinking if theres a more fundamental issue that could be addressed elsewhere.
6. I've not had much luck teaching people to change behaviour by directly asking, e.g. 'you should generally write more tests'. I find it more suitable to comment on the exact pull request only, rather than try to generalise. E.g. 'It looks like this could use more tests around xyz scenarios'. People tend to notice if you ask them the same thing for 4-5 pull requests in a row, and will generally be thankful you can maintain patience. Pull requests are often public so it's dangerous to issue any generalisations as people may get embarrassed or feel attacked. To save face, a private message or quick face to face chat about testing strategy may work if behaviour needs addressing.

7. I like to have links handy to common anti patterns and code smells, for the basics such as long functions (https://sourcemaking.com/re.... This can help remove personal opinions, give more weight to your suggestions, and also enable the submitter to read more, learn around the subject without you needing to explain things.
8. I try not to review the same pull request more than 2 times. If they've come back with a new pattern, and then again, generally I admit its good enough and let it through. Similar to how you say 'dont jump in front of every train' I would say 'don't hold the train up for too long'.

Reply

Philipp Hauer 4 years ago

Hi Michael,
you have some really good points! You seem to have reflected a lot about this topic. That's great. :-)

I like the way of expressing your opinion via the team perspective. That should work most of the time, but at the end, you are assuming the team's perception of the reviewed code. That is something that the author can argue against ('I don't think that the code is hard to understand for the rest of the team'). But he never can argue against your own impression of the code. That why I would always argument form my own point of view.

Most of the time, we do our code reviews face-to-face (especially within the team). That's why most of my points refer to that situation. But you are right, doing code reviews online via PR requires a slightly changed approach (like be carefully with public and general criticism).

Regarding the 'you should generally write more tests': You are right, I also wouldn't write that in a public pull request. It was just an example to show the difference between criticizing behavior vs attributes. In reality, I would do it as you pointed out: Refer to the concrete code that lacks test coverage.

Reply

Valentin PARSY 4 years ago

Awesome ! As someone that has trouble communicating as a reviewer I'm sure this will help a lot \o/

Reply

Philipp Hauer

I am Philipp Hauer and I work remotely as a Head of Engineering for commercetools in Leipzig, Germany. I strive for building happy, motivated, and productive teams and I'm enthusiastic about Kotlin, clean code, distributed systems, testing and the sociology of software development. I'm tweeting under @philipp_hauer, giving talks and offering leadership coaching, consulting and workshops.

‹ Install Cairo and CairoSVG on an Alpine Docker Image
Kotlin and MongoDB, a Perfect Match ›

Recent Posts

Categories

Top Tags