Code review guideline

Benefits of code review

The benefits of code reviews are plenty:

- someone spot-checks your work for errors
- they get to learn from your solution
- the collaboration helps to improve the organization's overall approach to tooling and automation.
- it will make everyone a better engineer because they will learn about the code of the other developers and give an opportunity to share experience and enhancements.

What is covered by code review?

Code reviews look at the change itself and how it fits into the codebase. They will look through the clarity of the title and description and "why" of the change.

What is cover:

- The correctness of the code
    - Does the code function as I expect it to?
- Maintainability observations
    - Do I understand what the code does
    - Complex logic
    - Hard to understand code
    - Unclear names
    - Commented out code
- Test coverage & improving test structure
- Functionality changes / "Why" of the change
- Confirm that they follow the coding guides and best practices.
- Unhandled edge cases
- Other possible improvements
    - Performance

Who should do a code review?

Every developer in the project should do a code review. It's important to involve the team in this process.

It's not recommended to have a gatekeeper for code review. But sometimes, this role can be added temporarily to coach the team about how to do a code review.

Review best practices

Build & tests before Code Review

It's important that the code **build and tests** run without error before doing a review.

Must be small

It's easier to find defects when there is **less than 400 line** of code (LOC) at time OR when the PR take **less than 15 minutes** to review.

It's a difficult cognitive process to review too many LOC / files.

Try to **break down the changes** into more focused parts and represent **single purpose** changes.

Give feedback that helps

The tone of code reviews can greatly influence morale within teams.

Never forget that the person writing the code spent a lot of time and effort on this change.

Embrace positive tone

A professional and positive tone can contribute to a more inclusive environment. People in these environments are open to constructive feedback and code reviews can instead trigger healthy and lively discussions.

Avoid harsh tone

Reviews with a harsh tone contribute to a feeling of a hostile environment with their microaggressions. Opinionated language can turn people defensive, sparking heated discussions.

Best practices

- Ask open-ended questions instead of making strong or opinionated statements
- offer alternatives and possible workarounds that might work better for the situation without insisting those solutions are the best or only way to proceed
- Be empathetic
- Assume the reviewer might be missing something and ask for clarification instead of correction.
- Know that the person writing the code spent a lot of time and effort on this change. These code reviews are kind and unassuming.
- Applaud great solutions and be generally positive.

Should be reviewed shortly

Try to review the pull request in one business day.

Review checklist

☐	**Pull request must have description**	• Does the PR provide a contexte and a description that explain the reason behind the code changes?
☐	**Readability / Understandability**	• Code is it easy to understand? • Terms, verbs describe correctly what the code is doing? • Is it easy to discern the role of specific functions, methods, or classes? • Code is break in small parts / by responsability? • Code not have too much statement? • Code not contains magic number • Repect naming convention • Method name should have only one a verb; When the method has more than one verbs, this indicates does more than one thing.

☐	**Maintainability**	• Respect coding guideline
		• Ensure that a method has not too much line of codes / it fits easily on your screen without scrolling.
		  • If not, it's probably that the method has too much responsabilities
		• Avoid inline comments. In most of cases, this reveal a code complexity. Favor a self explanatory code
		• Avoid method name that contains more than one verb
		  • Verb is related to a responsability/intention of the code
		  • Too much verbs reveal too much responsability
		  • See SOLID principle
		• Ensure that respect KISS principle (Keep it simple stupid)
		• Encapsulate complex expression in a method or property
		• Avoid nested loops

☐	**Security**	• Ensure the code does not constain vulnerabilities See OWASP top 10 (SQL injection, cross-site scripting, etc) • Are authorization and authentication handled in the right way? • Is sensitive data like user data, credit card information securely handled and stored? • Does this code change reveal some secret information like keys, passwords, or usernames?
☐	**Reliability**	• Don't use abbreviations, prefer explicit terms • Avoid using number in variables, parameters or types. In most of cases, they not reveal the real intention behind the name. • Should be easy to read for other developers • Avoid magic numbers because they not explicit • Avoid long code instruction on the same line

☐	**Reusability**	• Does the code use the existing treatment / library?
		• Does the code use appropriate language feature to accomplish the treatment?
		• Identify repetitive code using the rule of 3
		• Ensure that implement the right OOP concepts
		  • Encapsulation
		  • Abstraction
		  • Inheritance
		  • Polymorphism
		• Promote refactoring over copy/paste

☐	**Consistency**	• Use the same term always explain the same concept • Respect the nomenclature and the naming conventions • Have a consistent indenting and spacing • Use standardised architecture and interface
☐	**Fit for Purpose**	• Code may work, but does it work in the way that your Product Manager, CEO, or the user expects? Before code is pushed to production, it's worth double-checking that the code actually provides the functionality it was meant to provide.

☐	**Test Coverage and Test Quality**	• Ensure that unit tests are created related to new codes
		• Ensure that the code coverage is respected
		  • Add the expected code coverage into the DoD (definition of done)
		  • **Static code analysis** can help to identify and block this situation
		• Ensure that the naming convention is respected and are consistent
		• Ensure that the name reveals what the test does
		• Avoid repetitive code in unit tests.
		Unit tests are code. You should respect all OOP and basic principles than all other good coding practices.

☐	**Documentation**	• Does the code require documentation?
☐	**Speed and Performance**	• Do you think this code change will impact system performance in a negative way? • Do you see any potential to improve the performance of the code?

References

- https://medium.com/geekculture/the-top-10-checks-you-should-do-before-raising-a-pull-request-6dea166515ac
- https://medium.com/@mohsho10/c-best-practices-and-code-review-checklist-25880d9606
- https://blog.devgenius.io/why-code-consistency-is-important-9d95bdebcef4

- https://stackoverflow.blog/2019/09/30/how-to-make-good-code-reviews-better/
- https://smartbear.com/learn/code-review/best-practices-for-peer-code-review/
- https://www.perforce.com/blog/qac/9-best-practices-for-code-review
- https://www.codementor.io/blog/code-review-checklist-76q7ovkaqj
- https://github.com/mgreiler/code-review-checklist