Engineer

Code quality

how well code works, reads and can be changed

What it is

Code quality is how well source code implements its intended behavior and how readily people can understand, verify and safely change it. You can see the difference in a small pricing function. One version produces the right value on an ordinary order because it happens to follow the same sequence a designer demonstrated. Another expresses the rounding rule in one place, identifies the currency and handles refunds explicitly. Both may pass a happy-path test. The second gives someone investigating a disputed invoice a clear place to start.

Correctness includes boundaries such as invalid input, retries and partially completed work. Clarity concerns the concepts and choices the next reader can recover from the code. Changeability is about how much a later requirement disturbs and whether the team can observe the effects. These properties overlap. A complicated function can be correct today while leaving the next currency change difficult to verify. A short, readable function can still calculate the wrong tax.

Code review is one practice teams use to assess these properties. A quality gate is a rule that can stop work from advancing when selected checks fail. Technical debt describes future costs created or accepted through a choice. Code quality is the condition of the code itself. Google's review standard asks reviewers to consider whether an individual change improves the health of the codebase as a whole. That gives the broader property an everyday point of application.

Use the concept in a decision

Put "What is code quality?" to work in a real decision. We can help relate the concept to the system you are building or buying. One team taking an idea from a sketch to something your customers can log into, instead of handing it between three suppliers.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Led a GIS map application to 10x revenue growth — 10x revenue growth, led from the map product

Led a GIS map application to 10x revenue growth Get in touch →

Why it matters

A team experiences code quality most clearly when a requirement changes. Take a service that computes a quote, creates an invoice and later processes a refund. If each operation independently calculates a discount, changing eligibility means finding and updating every copy. A missing copy may show up only when one customer requests a refund. If the service applies a documented rule at a shared boundary and each caller supplies its own inputs explicitly, the reviewer can map the change and predict where to test it.

Useful evidence begins with the risk. For money, test the rounding boundary with examples where rounding each line comes to a different total from rounding the invoice once. For an operation that can be retried, record whether a second request can trigger a second charge. For a calculation that rarely changes but runs during every checkout, confirm the load and the required response time before adding caching. Each observation supports a judgment within a particular boundary.

The effect is also social. Engineers rotate, hours of availability change and a reviewer may never have worked with the original author. Clear names, an executable example and a short explanation of an unusual constraint allow another person to verify the important detail independently. That ability matters during incidents and handovers as much as during planned development.

What it looks like in practice

Consider a hypothetical invoice rule where the business rounds tax on the total for each tax rate. On a two-line invoice the unrounded tax on each line is 0.005 monetary units. Rounding each line half up to cents gives 0.02 in total. Adding first and rounding once gives 0.01. The example is deliberately small so a reviewer can settle the arithmetic without a spreadsheet. It illustrates a choice that must come from the product and applicable rules for the real invoice.

Start by naming the unit and reading order in the function contract. Keep intermediate values in a representation that preserves the required precision. Make the grouping step visible and maintain one place for rounding at the

agreed boundary. The test should use example values with a known result and state why they distinguish the two rules. Choose a second test for mixed tax rates so a future refactor cannot replace grouping with one aggregate. Give any separate refund policy an equally explicit example.

Review the smallest change that repairs the rule. Read the customer-facing amount, the exported amount and the caller that initiates the refund. Check that historical records retain the amounts stored at issuance. Exercise the full invoice path once to verify the callers actually pass the right units. A focused test can answer whether an isolated calculation is correct while the end-to-end check shows the calculation was hooked up at the right point.

Our note on selling the last ticket exactly once shows the same evidence-first method for a different domain. It writes down the inventory invariant, outlines concurrency tests and proposes watching the invariant in operation. The formula tells a reviewer what the implementation must preserve across several paths. Put that sort of condition where engineers can read it and verify it.

Where it goes wrong

First, a team may mistake a measurable signal for a result. Coverage shows which statements or branches a test executes. It cannot tell whether the expected result encodes the correct rounding policy. A complexity score can point to code worth reading, but replacing one obvious comparison with several obscure helpers can make an incident harder to diagnose. Ask which concrete defect a new measure would help the team catch and read representative examples where it passes and where it fails.

Second, completeness of a cleanup can become the wrong target. A confusing helper used by a single low-risk report might wait. Unclear logic used by every payment or by an irreversible migration has a different cost. Record the trade-off, the observed failure or exposure and when to look again. During a review, separate required correctness changes from optional improvements. A style dispute belongs at the level of a shared convention and can often be settled by an automatic formatter.

Third, a local improvement can move a problem downstream. Centralizing discounts makes sense where several callers share one business rule. It becomes misleading if renewal, promotion and refunds follow genuinely different rules but the helper guesses which one applies. Start by describing which inputs and

invariants really match. Share only the stable part and express the differences explicitly. The test suite and the next person's ability to explain the code should both improve.

Further reading

- Google's code review standard
- What Google reviewers look for
- Software Engineering at Google on unit testing

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how "What is code quality?" applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/code-quality/