Engineer

Regression testing

checking that a change preserves existing behavior

What it is

Regression testing checks whether software still performs previously correct behavior after a change. A new export setting can work exactly as specified and still alter the rows downloaded by existing users. Testing only the new setting answers whether the requested capability works. A regression test exercises the old path and asks whether a behavior people rely on still holds.

The NASA Software Engineering Handbook discusses regression testing as a way to confirm that changes have not introduced new defects. Its guidance includes re-executing existing tests that trace to modified code or requirements and choosing a scope that covers interacting modules. The underlying idea applies whether the change affects source code, a dependency, configuration, data or the environment. The same code can produce a different result when an input or browser changes.

Verification of a defect fix and regression testing work together. First reproduce the problem and confirm the new result after the fix. Then test nearby behavior that must remain correct. The case for the original fault can remain in the suite so it warns if the defect returns. The expected result should describe the behavior the team promises. When requirements deliberately change, update those expectations and document which existing callers have a different path.

Use the concept in a decision

Put “What is regression testing?” 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

Software changes rarely stay inside the visible edit. Turning an internal list into a set can remove duplicates, which may solve one problem and reorder a report. A date parser change can accept the new format and drop support for empty values handled elsewhere. An automated regression check catches particular effects while the team can still connect them to a specific change. This matters especially for behavior used by another integration that the author rarely runs locally.

For a small change, start with the nearest shared rule and one or two realistic different callers. Ask how the change could disturb default behavior, boundaries, error handling or already stored data. For higher-impact changes, consider effects across the product, such as preserving key performance and accessibility behavior as well as values. Select checks whose evidence is good enough for the consequence of a miss. A stable unit test can cover a conversion precisely. A browser check might be needed to see the actual visual or keyboard behavior using the same path as a reader.

Recorded examples help maintainers understand why a rule exists. A test called “empty date stays empty in an export of older cases” says more than “fixture three passes”. Show the important inputs and the relevant result. The next person can decide whether a new requirement intentionally changes the judgment or whether a failure signals something previously used has broken.

What it looks like in practice

Imagine a CSV export with a new “include archived accounts” option. The default export must keep excluding archived accounts. Active accounts remain sorted by account reference, and a missing last-contact date stays empty. The new option must add archived accounts without duplicating active rows. Give the fixture two active accounts in reverse order and one archived account. Leave the date blank for one active row. Exercise the old and new export paths through the actual exporter.

On the default path, compare the account references and their order, and confirm the empty date cell remains empty. With the option on, confirm the same active accounts plus the archived record, each exactly once. Include a name containing a comma, because an exporter that builds rows by concatenation can still return the correct row count while producing the wrong columns. Parse the generated CSV with the kind of reader a caller uses. The fixture should be small enough to read and representative enough to expose the effect.

Now check whether the tests reject a real regression. Temporarily make the default include archived accounts, or remove quoting for the comma. The relevant old-path checks should fail for distinct, readable reasons. Restore the implementation and run again. That exercise checks that a misleading assertion such as “the output file exists” did not quietly replace comparison of the records. Testing another known case with no archives helps preserve the valid empty selection path.

Our note on Safari underlines and font subsets offers a different example of evidence boundaries. It explains that a CSS change appeared to repair Chromium’s rendering while Safari still drew ink across letter descenders. A regression suite for the affected font behavior would need observations in the engines and cases it claims to protect. The note documents the diagnosis and browser evidence. The proposed suite is an example of how to carry the lesson forward.

Where it goes wrong

One trap is accepting test count as coverage. Hundreds of simple assertions can miss the only caller that reads historical data. Choose cases from the behavior inventory. Start with what real users do, which inputs exist and which interactions the change disturbs. When an issue escaped before, protect the workflow that suffered and check that the assertion fails when the faulty behavior returns.

Another trap is comparing too much. Full snapshots can change when file order, random identifiers or time vary. If the export promises the order of records, assert that order. If it promises only a set of records, compare the set while separately detecting duplicates. Test the quality the user depends on. Review any intentionally updated expectation as carefully as a code change and retain a reason for it.

Finally, a flaky check makes repeated failures hard to interpret. Replace timing guesses with controlled waits for observable conditions, use known test data and ensure one example does not leave state for the next. A test still needs a time bound and a readable failure. Keep broader checks for the points where genuine interactions matter, and keep quick focused checks close to development. The combination gives the team evidence about different parts of the same change.

Further reading

- NASA Software Engineering Handbook on software regression testing
- pytest documentation on parametrizing examples
- Software Engineering at Google on effective unit tests

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 regression testing?” 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/regression-testing/