Engineer

Technical debt

the extra cost later of a quick, expedient choice made now

What it is

Technical debt is the future cost of a shortcut in the design or code of a system: a choice that saves time today and makes later changes slower, riskier or impossible until the shortcut is undone. Teams often shorten it to tech debt. The word is a metaphor, and it was chosen because the shortcut behaves like a loan. It buys something now, and it charges for it later, for as long as it stays in place.

The idea comes from Ward Cunningham's experience report at OOPSLA in 1992, on WyCash, a portfolio management system written in Smalltalk. A small team kept general knowledge of roughly four megabytes of source, and the mature parts of the program had been revised or rewritten many times. That consolidation, the report argues, is what made steady incremental growth possible, and its absence is the serious pitfall: code that works and pleases the customer, but has never been brought into shape. Cunningham put it in one line: "Shipping first time code is like going into debt." The report itself speaks only of debt and of its debt analogy. Later work uses the two-word name, and a 2015 paper from Google credits the metaphor to Ward Cunningham and dates it to 1992.

Researchers who met at Dagstuhl in April 2016 to agree on what the term covers wrote a working definition. In it, debt is a set of design or implementation constructs that save effort in the short run but leave the system in a shape that makes later changes cost more, or puts some out of reach. They limit its impact to qualities inside the system, above all how easy it is to maintain and to evolve. That separates debt from a feature nobody built: the cost of debt falls on the people who change the system, and reaches users only through them. The same report keeps the financial vocabulary. The principal is what the shortcut saved, or what a better solution would cost today. The interest comes in two kinds: recurring interest, paid as lower velocity, induced defects and lost quality, and

accruing interest, the extra cost of every new piece of software built on top of the code that is not quite right. Debt lives mostly in code, and also in documentation, tests and known defects.

Use the concept in a decision

Put “What is technical debt?” to work in a real decision. We can help relate the concept to the system you are building or buying. The shape of a system worked out on paper first, so the parts fit and the cost is known before anyone writes code.

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. One infrastructure plan for 14 departments — 14 research departments on one infrastructure plan

One infrastructure plan for 14 departments Get in touch →

Why it matters

Debt taken on deliberately can be a good decision. Cunningham’s report says a small amount of it lets a team move faster, provided it is settled soon by rewriting the code, and the Google paper agrees that borrowing is often a sound strategic choice. What both warn about is debt left in place. In the report, the interest is paid in time, with each minute lost to code that is not quite right, and an implementation nobody consolidates can grow heavy enough to halt whole engineering organisations. Before that point, the program becomes something no single person can master, which leads to programmers who each know one corner and finally to a product that cannot change direction.

The trouble is that debt is often not chosen. The Dagstuhl report observes that it often arises without anyone meaning it to and takes developers by surprise, which is why it is mostly discovered during maintenance, long after the decision that created it. Its causes are ordinary: schedule pressure, the absence of a key person, or too little information about a technical feature at the time. The Google paper adds that hidden debt is the dangerous kind, because it compounds silently.

The metaphor also gives developers and managers a shared language. The Dagstuhl report calls it a strong communication mechanism, and the vision its participants set out includes managing debt as carefully as defects and new features, with a way to translate between what developers worry about and what managers decide. A study presented at the seminar shows what that can mean in a team. Graziela Tonin, Alfredo Goldman and Carolyn Seaman asked student teams of 60 people in total to identify and track their debt explicitly. A

visible list let teams negotiate time with their customer for refactoring, kept a record of which parts of the project were still immature, and showed whether code quality was improving. Most team members said they communicated more, thought harder about whether a debt was really needed and refactored more often. Because decisions to take on debt were talked through, the authors add, some teams ended up with fewer experts nobody else could stand in for.

What it looks like in practice

Debt becomes manageable when it is written down where the team plans its work. An entry in a register, written for this article, can be as short as this:

```
TD-14       Invoice totals are calculated in three places
Artifacts:  billing/web, billing/export, reports/monthly
Cause:      a fixed launch date; copying the rule was quicker than
            moving it into a shared module
Principal:  about 2 days to move the rule into one module, with tests
Interest:   every tax-rule change is made and tested three times, and
            one of the last five changes missed a copy
Next:       repay with the next tax-rule change; the e-invoice export
            planned this quarter must not become a fourth copy
Owner:      billing team
```

The fields follow the model from the Dagstuhl report: an item tied to concrete artifacts, with a cause, a principal and interest. Writing the interest down is what makes the entry useful, because it turns a feeling that the billing code is messy into a cost someone can weigh. If tax rules change four times a year and each change costs an extra day of work and testing, the two days of principal are repaid within six months. The planned export is accruing interest in the Dagstuhl sense: every new piece built on the duplicated rule raises the price of fixing it later. If, on the other hand, the code were frozen and nobody expected to touch it again, the same entry would justify leaving the debt alone.

Debt is not only a matter of code. This site’s portfolio entry on overhauled internal processes, saving 8,000 hours records the founder’s earlier work at Mappitall. It asks whether technical debt is slowing the reader down, and describes the kind that builds up around how the work runs: software architecture that had grown by accretion rather than design, systems that worked but not efficiently, and scheduling that left people either waiting or overloaded. None of it was on fire, which is why it had been left alone. Reworking the architecture, the systems and the scheduling saved roughly 8,000 hours over

the work carried out from 2015 to 2022, and the page credits part of that to making the changes stick in how the team operated, since improvements that are not made permanent decay back to the old way.

Where it goes wrong

Everything is called technical debt. Once the phrase is popular, every annoyance gets the label, from an old library to a missing feature to a colleague's style. The Dagstuhl report names this risk directly and asks for crisp definitions in answer. Keep the term for shortcuts whose cost lands on future change, record defects as defects and wishes as features, and refuse a register entry that cannot name its cause and its interest.

The debt exists only in people's heads. Debt nobody wrote down is paid for by whoever touches the code next, usually without knowing why the work takes so long. Keep a register next to the backlog, and check it before changing a part of the system, the way the student teams in the Tonin study used their list. An item becomes visible to the people who decide on time only when it is written where they look.

Principal is repaid where no interest is charged. Cleaning up a module that nobody changes feels productive and returns nothing, while the code that changes every week keeps charging. Interest only accrues where people work, so rank the register by how often each part changes and by what is about to be built on it, and repay the expensive items first.

Repayment waits for one large rewrite. Debt left for years tends to be answered with a project to replace the whole system, which takes on new debt of its own while the old system keeps charging. Cunningham's advice was prompt repayment, and the Google paper lists the usual ways to pay it down: refactoring, stronger unit tests, removing dead code, fewer dependencies, tighter APIs and better documentation. None of these adds a feature; the point is to make the next change cheaper. Schedule them as part of ordinary work, close to the change that needs them.

Experiments stay in production. Trying an alternative as a conditional branch in the production code is cheap for each experiment, and the Google paper describes how such branches pile up until testing every combination becomes difficult or impossible. It cites Knight Capital, whose system it says lost $465 million in 45 minutes, apparently because obsolete experimental code paths

behaved unexpectedly. Re-examine each experimental branch and each flag on a schedule, delete what is no longer used, and reward deleting code as much as adding it.

A tool's score is taken for the debt. Static analysis can count duplicated lines and long functions, and a dashboard makes the count look like a balance. The Google paper is plain that the metaphor provides no strict metric to track over time, and that a team still moving quickly is not evidence of low debt. At Dagstuhl, Forrest Shull reported that no single set of rules for detecting debt correlated consistently with its symptoms, such as rising change cost and falling velocity, across different projects. Check a tool's findings against your own history of changes and defects, keep the rules that predict cost, and drop the rest. The paper's own questions are a better test, among them how quickly a new member of the team can be brought up to speed.

Further reading

- The WyCash Portfolio Management System — Ward Cunningham's OOPSLA '92 experience report, where the debt analogy first appears.
- Managing Technical Debt in Software Engineering — the report of Dagstuhl Seminar 16162, with the working definition, principal and interest, and the study of explicit debt lists.
- Hidden Technical Debt in Machine Learning Systems — the 2015 paper by Sculley and colleagues at Google on how debt builds up, how to pay it down, and why it resists measurement.
- Refactoring — Martin Fowler's site on the small, behaviour-preserving changes that the Google paper lists first among the ways to pay debt down.

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 technical debt?” 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/technical-debt/