Creating Tech Debt Ownership culture

Introduction

In every team, sooner or later, someone proposes a process to manage technical debt, a spreadsheet, a Jira tag, a weekly review, or an ADR template, mentioning some.

These are good tools. They help organize and track what matters. But we often miss the point, because the real challenge with technical debt isn't the absence of a process, but the absence of a shared culture.

The problem isn't what happens when you follow the steps. The problem is what happens when no one is looking.

If we want to build scalable, maintainable systems, we need more than tools. We need to build a mindset. A sense of ownership. A commitment to the product. A team that cares.

"Small leaks sink great ships." –*Benjamin Franklin*

Table of Contents

01 Why Culture, Not Just Process

1 Why Culture, Not Just Process

Technical debt is not just old code, or missing tests, or that quick fix nobody remembers writing. It's a symptom.

A symptom of rushed decisions, of unspoken assumptions, of postponed conversations. And ultimately, it's a reflection of how much (or how little) we care about the product.

A process alone can't solve that.

A process tells you what to do once something is detected. However, curiosity, creativity, and attention to detail (those things that help notice what's wrong in the first place) are not process-driven. They're cultural.

- Culture is what makes a developer stop and say, “This seems slower than usual.”
- Culture is what makes someone log an issue when a workaround starts to become the default.
- Culture is what encourages the question: “Does this still make sense?”

Processes rely on visibility and structure. Culture lives in the invisible moments. The hunch. The small refactor. The documentation was left for the next developer.

And here's the truth: as tech leads or engineering managers, we are not present in those invisible moments. We can't be. We're not there for every PR, every commit, every workaround as developers do.

That's why we need a team that thinks and acts with ownership, because no process will ever be as powerful as a developer who cares enough to ask: “Is this the best we can do?”

02 The Role of the Tech Lead

From Gatekeeper to Gardener

2 The Role of the Tech Lead: From Gatekeeper to Gardener

If culture is the soil where good habits grow, then tech leads are the gardeners.

Our job is not to control every commit or enforce every rule. Our job is to ***cultivate the conditions where care, curiosity, and responsibility can thrive.***

We don't build culture by writing more documentation. We build it by showing what we value, every single day.

2.1 Care about the code like any other developer

You can't ask your team to care about the code if you don't.

Even if you're not coding full-time anymore, you can show care in how you read a PR, in the questions you ask, in the bugs you investigate.

When the team sees that the tech lead still thinks deeply about the system, not just the roadmap, it sends a message: this code matters.

2.2 Invite, don't impose

People don't engage with processes they don't feel part of.

That's why building a culture of technical ownership starts with **invitation**. Share your concerns. Ask for opinions. Listen. Encourage suggestions, even when they're not polished.

Culture grows from participation, not compliance.

2.3 Keep your word

Nothing kills trust faster than saying, “We’ll fix this later” and never following through.

If you say you’ll come back to something, do it. If you promise space in the sprint for tech debt, make sure it happens.

Your consistency builds the team’s belief that improvement is possible.

Inconsistency builds cynicism.

2.4 Spread love into the code

We talk a lot about user experience, but not enough about **developer experience**. The code we work on shapes how we feel. Clean code feels good. Understanding things quickly feels good. Naming something well feels good.

Encourage your team to make the code more expressive, more human. To leave behind comments that help. To improve things not because they have to, but because they can.

This is what it means to love the product:

Not just the features, but the foundation.

2.5 See the bigger picture

Every technical decision is a trade-off.

Sometimes we take on debt to move faster. Sometimes we avoid it because the cost is too high. But what matters is that we know what trade-offs we’re making, and WHY.

Help your team connect their choices to the broader system, to business goals, to future maintainers.

Context turns reactive developers into strategic ones.

2.6 Listen to what the team is telling you, especially when they don’t say it directly

That weird silence during the sprint review? That “I wouldn’t touch that file” joke? These are signals.

Learn to read them. And when you hear concern about a decision, a tool, or a workaround, don’t ignore it.

Dig in. Ask questions. Culture is built in those conversations.

2.7 Believe in the 1%

Systems rarely break overnight. They degrade gradually — 1% at a time.
One skip in a test. One unclear variable. One temporary fix that overstays its welcome. Just like deterioration compounds, so does care. It's the Broken Windows effect.

One thoughtful refactor. One clarified use case. One doc that saves someone 10 minutes works the other way.

Leadership is believing that the 1% matters. In both directions.

03 The Daily Mindset

How Developers Keep Systems Healthy

3 The Daily Mindset: How Developers Keep Systems Healthy

Culture (as technical debt) doesn’t grow in meetings or processes; it grows in the tiny decisions people make **when no one is watching**.

The way developers think when they open a file. The questions they ask when something feels off. The choice between hacking something together or stopping to ask: “What’s the right way to do this?”

Here are some of the mindsets we try to cultivate every day — not as rules, but as instincts

3.1 Follow the Boy Scout Rule

“Leave the code better than you found it.”

It doesn’t have to be big. Rename a confusing variable. Delete unused logic. Add a missing document.

These small actions create momentum. They make it easier for the next person to care.
They say: someone was here, and they gave a damn.

Clean code always looks like it was written by someone who cares – Robert C. Martin

3.2 Ask the obvious questions

- Why does this SELECT take 3 seconds?
- Why do we have two flags doing the same thing?
- Why is this config hardcoded in three places?

Many code smells persist because people assume someone else has already asked. But they often didn’t. And even if they did, asking again is how we evolve.

Don’t normalize weirdness. Curiosity is a feature, not a risk.

3.3 Pause and check for drift

It worked, yes. But does it still make sense?

Codebases drift. A workaround from last year may no longer be necessary. A naming convention might be outdated. A “temporary” flag might have become permanent.

Just because something exists doesn’t mean it’s right. Healthy systems need people who notice when the map no longer matches the territory.

3.4 Write down your doubts

You don’t need to fix everything today. But if something feels off, say so. Leave a comment. Open a GitHub issue. Add a note in the debt tracker.

Your future self — or your teammate — will thank you.

3.5 Be calm enough to refactor

In high-pressure environments, it’s easy to slip into survival mode: just make it work.

But that’s when refactoring matters most, because it reduces friction for the next sprint. It makes the next hard thing less hard.

Refactoring is not indulgence. It’s resilience engineering.

04 Not All Tech Debt Is the Same

Learn to Name It Before You Fight It

4 Not All Tech Debt Is the Same: Learn to Name It Before You Fight It

Calling everything “tech debt” is like calling every injury a “bruise”; it oversimplifies, hides the root cause, and makes it hard to treat.

To address debt properly, we need a shared language. We need to ***understand what type of debt we’re dealing with*** and how it got there.

Based on the paper ***“Towards an Ontology of Terms on Technical Debt”***, we can identify multiple types of debt, each with different origins, consequences, and strategies.

Here are some common forms to be aware of:

4.1 Code Debt

The most visible kind: poor structure, duplicated logic, confusing naming, functions that do too much.

Easy to spot. **Easy to underestimate.**

Often grows from rushed timelines or missing standards. It can usually be fixed incrementally.

Habits for Developers:

- Apply the Boy Scout Rule: rename, extract, simplify, even if just a little.
- Avoid “just one more if”. Think in readability, not only logic.
- Apply a reverse **“The Principle of Least Surprise**”. Were you surprised? Change it or talk about it.

Habits for Tech Leads:

- Praise small refactors in PRs. Visibility builds culture.
- Normalize the time for cleanup — even in “feature” sprints.
- Create a safe space to raise concerns like “this part is getting too hard to maintain.”

4.2 Architecture Debt

When systems are hard to extend, scale, or modify because of deep design issues.

Symptoms include excessive coupling, missing boundaries, or components doing too many things.

This kind of debt is dangerous: it slows everything down and limits future options.

Habits for Developers:

- Ask yourself: Is this code doing more than one thing?
- Propose boundaries (even informally) when you feel things are too coupled.
- Pay attention to smells: too many flags, tangled conditionals, repeated logic across domains.

Habits for Tech Leads:

- Protect time for small redesigns before they become big rewrites.
- Track pain points: Where do features “get stuck”? Where does change feel expensive?

4.3 Documentation Debt

When decisions, dependencies, or assumptions are not written down.
Or worse, when they are, but no one trusts them.

This debt doesn’t block development directly. But it slows onboarding, reduces autonomy, and leads to repeated mistakes.

Habits for Developers:

- Don’t aim for perfect docs, aim for useful. A one-liner in a README is better than silence.
- When you have to ask a question, ask it and then document the answer.
- Use PR descriptions and commit messages as a place to explain reasoning, not just changes.

Habits for Tech Leads:

- Lead by example: write and update documentation yourself.
- Make “documentation is part of done” a consistent (and flexible) expectation.
- Recognize and reward effort spent on making knowledge accessible.

4.4 Strategic Debt

Sometimes, you choose to take on debt on purpose, to hit a deadline, validate a hypothesis, or unlock a business goal.

This is not bad. In fact, it can be smart. But only if you track it, revisit it, and eventually pay it back.

Strategic debt without a strategy to resolve it is just neglect dressed up as pragmatism.

Habits for Developers:

- When taking a shortcut, say it. Write it down to have a track.
- Be explicit: What did we compromise? Why? When should we come back to this?
- Push for visibility: silent shortcuts turn into invisible landmines.

Habits for Tech Leads:

- Be honest about trade-offs. Say when something is debt and explain WHY.
- Track strategic debt in the backlog or Tech Debt Sheet.
- Close the loop: revisit the shortcut later. Don’t let “just for now” become “forever.”

The rest can be revisited on the shared paper.

4.5 The Rest

There are other types of technical debt to be aware of. You should understand them and keep them present, as it will help us create or adapt processes once we create the right culture.

You can find more information in Towards an Ontology of Terms on Technical Debt.

05 Turning Culture Into Process

Building It Together

5 Turning Culture Into Process: Building It Together

Introducing a process to manage tech debt is not the first step, but the result of cultural change. It's how you formalize the mindset you build with the team. Once you have taken care of the field, the process is your crop.

Too often, processes are imposed from the top, or inherited from other teams, or adopted reactively after a problem. When that happens, teams see them as extra work, not as something meaningful.

But when a process is built with the team, not for the team, something different happens. It becomes a mirror of your values. A shared language. A space where improvement is intentional, not accidental.

A well-designed process is not a set of rules — it's a framework for shared thinking.

When we designed our technical debt process, we didn't start by filling a spreadsheet. We started by listening:

- What frustrated the team?
- Where did we keep stepping on the same landmines?
- What kind of debt was acceptable, and what wasn't?
- What would make it easier to improve things without asking for permission?

That conversation didn't just help us define the process; it helped us define what *we* cared about as a team.

This is what makes the difference: **ownership**. The moment when the process is not something that belongs to the manager, but something that belongs to the team.

06 One process

6 One process

The process I'm sharing was created for use on Loyalty, Antares, and Altair. And I built using the team feedback and my own background. Both things can be different for you, and you need to understand what things are valuable to you, your team, and your services; create a process that reflects that. Take this just as an example, and I hope you find it useful to build your own.

In this repository, you will find some resources, like "Towards an Ontology of Terms on Technical Debt", ADR templates, and our process itself.

https://github.com/trafilea/loyalty-resources