Managing Technical Debt

Povilas Balzaravičius, Software Engineer, Uber

UBER

August, 2016 Vilnius PHP

About Me

Povilas Balzaravičius

Software Engineer at Uber

I have over 10 years' experience as a software engineer on internet projects of various size. In early 2015, I joined Uber's Developer Experience team.

DevExp builds and maintains tools used by other engineers. Our mission: make being an engineer and doing the engineering awesome!

I'm also a proud father and husband, organizer of the first and only hacker camp in Lithuania (No Trolls Allowed), leader of Vilnius PHP user group, and active participant of other Lithuanian user groups.

Agenda

Managing Technical Debt

- Definition
- How Technical Debt Happens
- Impact
- Identifying Technical Debt
- Dealing with the Debt
- Outcome
- Q&A

Definition of Technical Debt

Financial Debt

Financial debt is not always a bad thing.

Lets us get things or services that we need, but cannot afford now.

Enables us to keep the same quality of life.

Financial Debt

But financial debt can be dangerous!

It needs to be managed by paying interest.

The debt hides problems and leads to other problems.

If interest and debt itself is not paid back, it leads to bankruptcy.

Definition of Technical Debt

The term was coined by Ward Cunningham in 1992.

Technical debt (also known as design debt or code debt) is a metaphor referring to the eventual consequences of any system design, software architecture, or software development within a codebase.

"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite…. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt.”

Ward Cunningham, 1992

How Technical Debt Happens

How Technical Debt Happens

Social pressure: deliver new features as soon as possible

Estimation: always wrong; we always underestimate

Projects last for a long time: decisions made three years ago aren’t right anymore

Project owners tend to evaluate software quality and performance by number of implemented features.

How Technical Debt Happens

- Unexperienced developers
- Faulty architecture decisions
- Incorrect technologies selected
- Lack of automation
- Lack of testing
- Lack of knowledge sharing

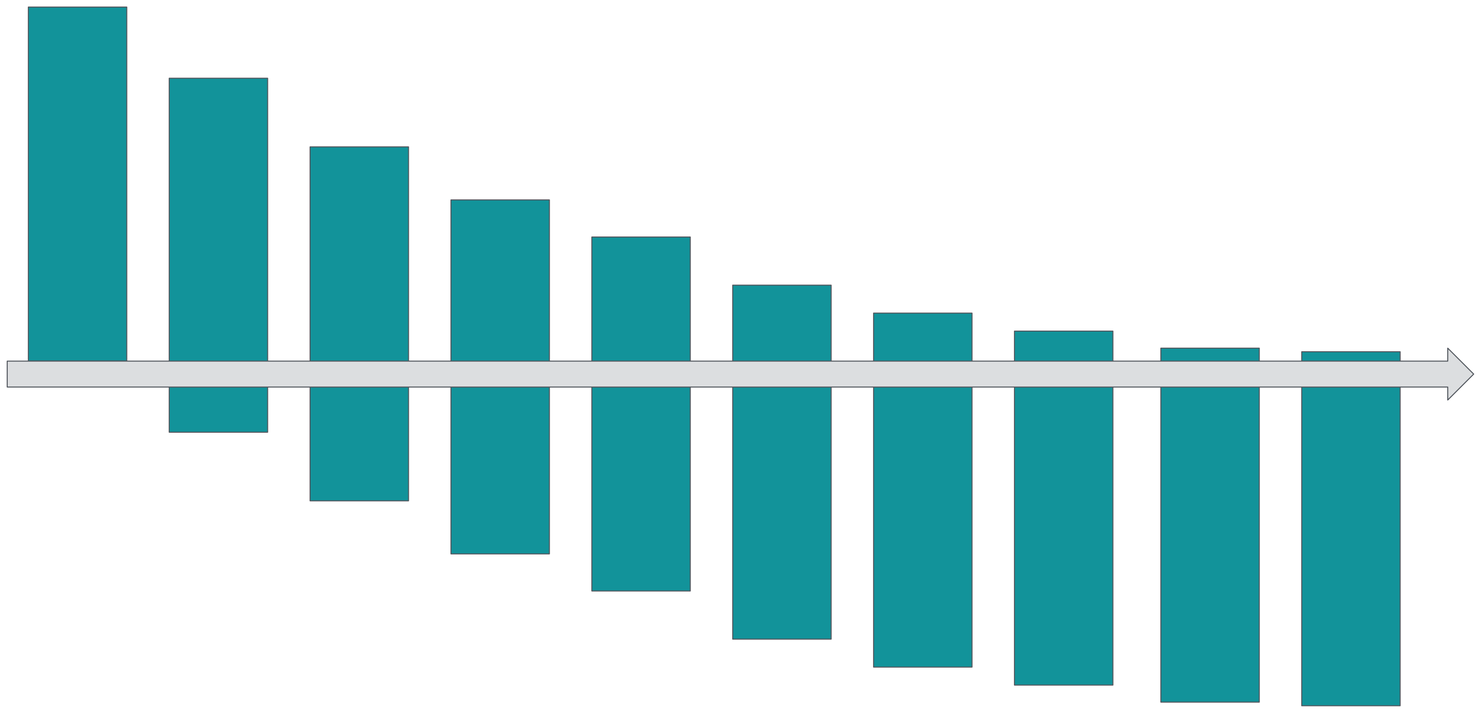
Shipping your first line of code is already a debt.

print “hello world”

Impact of Technical Debt

Time to Implement New Features

Example of Unhealthy Team



Time available to
implement new features

Time spent on debt

Impact of Technical Debt

Business

New features take longer to implement—at an increased development price.

You experience failures in software, especially after upgrades.

Impact of Technical Debt

People

Huge debt leads to frustration.

People leave because of frustration.

Most frustrated people are those who understand how quality software should be written.

Identifying Technical Debt

The main difference between financial and technical debt:

Technical debt has no balance sheet.

Identifying Technical Debt

You have debt if...

- Software uses super fancy, yesterday-released, on-hype technology
  - Database
  - Framework
  - Language

New technologies generally haven’t been stress tested well and have many bugs.

Authors tends to release new versions and change APIs often.

Pick the safest and cheapest-to-adapt technologies.

Identifying Technical Debt

You have debt if...

- Code structure is monolithic and components are highly coupled (related to each other)

Writing loosely coupled components and services forces us to define clear interfaces for how components must be used.

This lets us modify component behaviour or replace a component easily.

Separate teams can work on separate components.

Identifying Technical Debt

You have debt if...

- Code is duplicated
- Code has workarounds to cut paths

The willingness to deliver features faster sacrifices code quality.

This leads to more failures and more code parts that nobody wants to touch.

Identifying Technical Debt

You have debt if...

- Documentation is nonexistent, poor or even incorrect
- Methods, variables, and other code parts have incorrect or inconsistent naming
- Commented-out code

This makes code hard to understand, especially if documentation or names are misleading.

Identifying Technical Debt

You have debt if...

- Team manually performs same tasks before or after releases

Manual testing and other processes must be automated to prevent human error and save time for more releases.

Identifying Technical Debt

You have debt if...

- Test coverage is low
- You have no tests

Tests let developers know their changes work properly and haven’t introduced new failures.

Refactoring is also impossible if tests are missing.

This leads to manual testing, which is technical debt as well.

Identifying Technical Debt

You have debt if...

- Tests are broken, incorrect or too sensitive

Incorrect tests don’t identify the exact point of failure.

Too-sensitive tests break with changes to other code parts or data.

Developers waste time to clean up tests.

Identifying Technical Debt

You have debt if...

- Your software is older than X years

Software ages.

Old software uses old technologies and dependencies.

Support for old libraries is deprecated sooner or later.

Developers are forced to become maintainers of used tools or switch to equivalents.

Identifying Technical Debt

You have debt if...

- You’re postponing releases
- Productivity has decreased
- Number of bugs has increased

All these things result from points mentioned in previous slides and show poor software quality.

Dealing with Technical Debt

Project cannot be debt free with the same high level of quality across all components.

Paying Off the Debt

Stop creating new debt.

Clean existing debt regularly.

Devote a set amount of time to cleaning up existing debt.

Four hours per week or one hour per day is a good choice!

Make sure you have time to work on new features.

Make Debt Visible

Create technical backlog.

If debt cannot be fixed immediately, file a task.

Technical debt is difficult to measure. Make it clear and visible to everyone on the team: developers, managers, and product owners.

Define the Meaning of “DONE”

Task status should be marked as DONE when all requirements are implemented, tested, and deployed.

Whole team must understand the definition of DONE and follow the rules.

The Broken Window Theory

“One broken window, left unrepaired for any substantial length of time, instills in the inhabitants of the building a sense of abandonment—a sense that the powers that be don't care about the building. So another window gets broken. People start littering. Graffiti appears. Serious structural damage begins. In a relatively short space of time, the building becomes damaged beyond the owner's desire to fix it, and the sense of abandonment becomes reality.”

The Pragmatic Programmer: From Journeyman to Master, ISBN-10: 020161622X

Don't Leave "Broken Windows"

Don't leave technical debt unrepaired.

Fix each broken bit of code as soon as it is discovered.

If there is insufficient time to fix it properly - create a task.

Take some action to prevent further damage and show that you're on top of the situation.

Make the code just a little bit better after every touch.

Refactor code

Improve structure

Deduplicate code

Decrease complexity

Fix method and variable naming

Clean up unused code

Improve test coverage

Rewrite if necessary

Don't leave "broken windows" unrepaired.

Just make sure to follow the plan you set. Spend no more time than the hours per week you reserved for technical debt.

Automate your tasks

Automate everything you can to avoid wasting time on repeatable tasks.

Run tests, code style checks, and deployments automatically.

Use tools to automatically fix and warn about problems even before code is pushed to repository.

Preparing project for automated processes forces developers to write flexible code.

Monitor Your Debt

- Code coverage
- Code style
- Coupling
- Cyclomatic complexity
- Unused endpoints and code

Decide which metrics are most important for team.

Agree on code style.

Use tools to monitor those metrics.

Act if threshold is reached (e.g., do not accept code change to be merged to master branch if test coverage is below 90%).

Peer Code Reviewing

- Collective code ownership, transparency, discipline, and familiarity
- Feedback on design and code decisions
- Bug discovery

Performing code review for every change requires the whole team to work together, share knowledge, and learn from each other.

Peer code reviews are one of the most effective processes for boosting code quality and eliminating technical debt.

When Debt Is Too High

What can we do if refactoring is not possible?

Replace outdated part with something not perfect but better if possible.

Encapsulate old code and set up new code base over it. Forward requests from new system to old one.

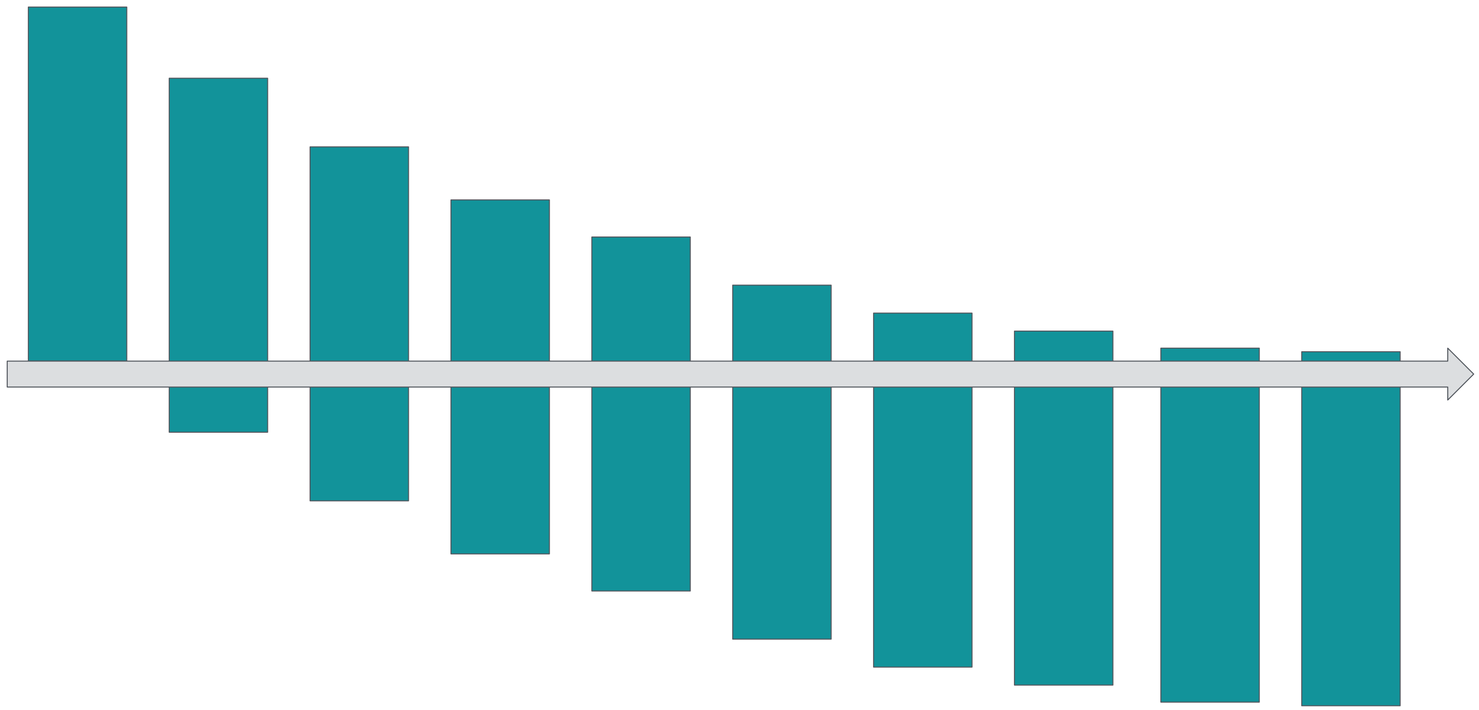
Sometimes refactoring is more expensive than working with the current code.

In those cases, just live with it.

Outcome

Time to Implement New Features

Example of unhealthy team

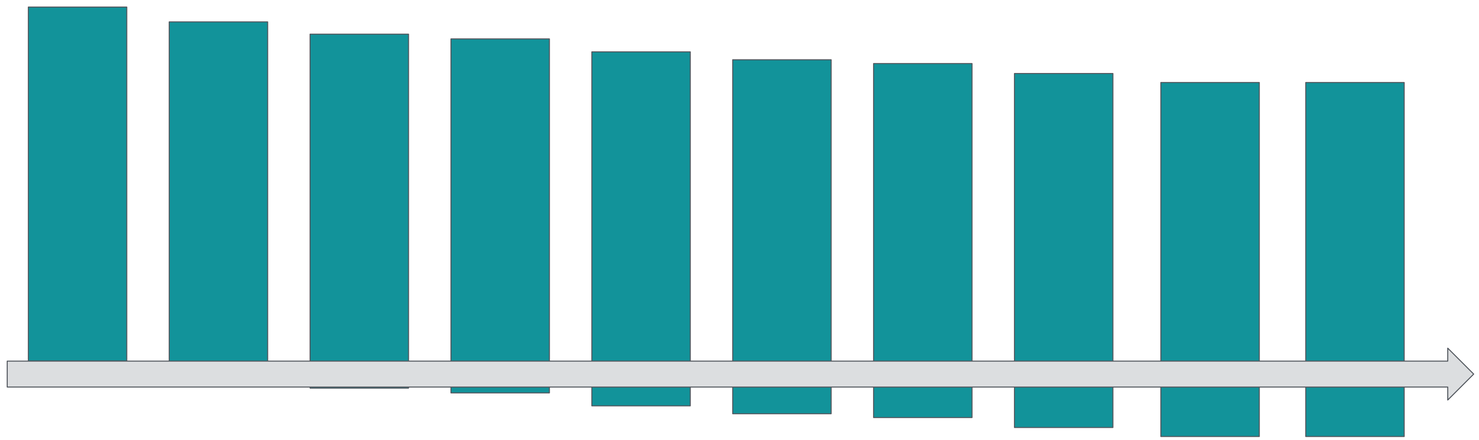


Time available to
implement new features

Time spent on debt

Time to implement new features

Example of **healthy** team



Time available to implement new features

Time spent on debt

Do not ask for permission to do your job properly!

Thank you

Povilas Balzaravičius
Software Engineer at Uber
+370 601 17345
povilas@uber.com