Technical Debt: Myths and realities

Philippe Kruchten

Philippe Kruchten

Professor Emeritus
University of British Columbia
Vancouver, BC Canada
pbk@ece.ubc.ca

Founder and president
Kruchten Engineering Services Ltd
Vancouver, BC Canada
philippe@kruchten.com @pbpk
Outline

• What is technical debt?
• The technical debt landscape
• Myths: Limits of the metaphor
• Realities: Practical steps

Slides will be at Philippe.Kruchten.com/Talks

Key takeaways

• All software systems accumulate technical debt, which is different than defects.
• How much technical debt you suffer from depends on the future evolution of the system, not just its past.
• While code-level debt is easier to identify and remediate, architectural debt has the highest cost of ownership.
• Being agile does not make you immune to technical debt.
Technical Debt

- Metaphor introduced by Ward Cunningham (1992)
- Until 2010, often mentioned, rarely studied.
- All experienced software developers “feel” it.
- It drags long-lived projects and products down

Origin of the metaphor

- Ward Cunningham, at OOPSLA 1992

“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. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.”

Cunningham, OOPSLA 1992
Technical Debt (S. McConnell)

- Implemented features (visible and invisible) = assets = non-debt
- Type 1: unintentional, non-strategic; poor design decisions, poor coding
- Type 2: intentional and strategic: optimize for the present, not for the future.
  - 2.A short-term: paid off quickly (refactorings, etc.)
    - Large chunks: easy to track
    - Many small bits: cannot track
  - 2.B long-term

McConnell 2007

“...A design or construction approach that is expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now.”

McConnell 2011
Technical Debt Definition 2016

In software-intensive systems, technical debt is the collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible.


Technical Debt Definition

Technical debt presents an actual or contingent liability that impacts internal system qualities, primarily maintainability and evolvability.
TD in your backlog: negative value, invisible

Visible
- New features
- Added functionality

Invisible
- Architectural, Structural features
- Technical Debt

Positive Value
- Technical Debt: negative value, invisible

Negative Value
- New features
- Added functionality
- Architectural, Structural features
- Technical Debt

Mostly Invisible
- Low internal quality
- Code smells
- Code complexity
- Coding style violations
- Test debt
- Documentation debt

Evolvability: resolution issues

Maintainability: quality issues

Copyright © KESL 2020
Code-level technical debt

- McConnell’s type 1 (and 2.a)
- Code smells
- Detected by static analysers
- Self-admitted technical debt

Causes

- Schedule pressure
- More schedule pressure
- Ignorance
- Success
- Environment evolution
  - Technical and business
- Sloppiness
Constant debt reduction

- Make technical debt a visible item on the backlog
- Make it visible outside of the software dev. organization
- Incorporate debt reduction as a regular activity
- Use buffer in longer term planning for yet unidentified technical debt
Types of Architectural debt

- The Minimal Viable Product (MVP) that stuck
- The Workaround that stayed
- Re-inventing the wheel
- Poor separation of concerns
- Architectural lock-in
- New context, old architecture

Cause of architectural debt

- Time pressure
- Lack of architectural knowledge
- Lack of architectural documentation
- Human factor: bias, ignorance, etc.
- The passing of time...
McConnell’s TD litmus test

• “If you are not incurring any interest, then it probably is not a debt.”

McConnell 2013

Interests

• In presence of technical debt, cost of adding new features is higher; team velocity is lower.

• When repaying (= fixing, remediating), additional cost for retrofitting already implemented features

• Technical debt not repaid => lead to increased cost, forever
• Cost of remediating increases over time

M. Fowler, 2009
Technical debt: Myths

- Ward Cunningham invented technical debt in 1992
- Technical debt is just another fancy name for defects
- Tools can identify technical debt incurred by a software system.
- We can measure technical debt incurred by a system.
- Technical debt is very bad and should be avoided at all cost.
- Agile teams are immune to technical debt

It’s a metaphor!
Metaphors

- Metaphors give meaning to form, help ground our conceptual systems.
- Cognitive transfer: source domain to target domain
  - the <target> is the <source>

_Lakoff and Johnson (1980) Metaphors we live by_

Where the mortgage metaphor breaks...

- Technical debt depends on the future
- Technical debt cannot be measured
- You can walk away from technical debt
- Technical debt should not be completely eliminated
- Technical debt cannot be handled in isolation
- Technical debt can be a wise investment
TD: Not a new concept

• Just a new term, exploiting a simple metaphor
• Technical debt existed before 1992
  – Low internal quality
  – Software evolution
  – Design erosion

TD /= defects

• The software does work.
• If it does not, call it a defect, and fix it (calling it technical debt is just a cop out)
• Technical debt does increase the likelihood of introducing defects => risk!

• Sometimes, but rarely, the boundary is uncertain.
Contrast:

Defect
- Visible
- External quality
- Function of the past only
- Not a good investment
- Multiple possible causes
- All systems

Technical debt
- Invisible
- Internal quality
- Function of past and future
- Can be an investment
- Mainly triggered by schedule pressure
- Large & long lived system

Bring me tools!

- Static analyzers will detect much of type 1 technical debt.
- More significant technical debt items (structure, architectural) cannot be detected by tools.
- Some team members know about them, though....
- They may be mentioned in discussions, but not visible in the code.
Measuring TD

• *To measure* is to assign a numerical value to an attribute of a thing

• Cost (Technical debt item) = ?

• Naively, the effort to bring the system to a state where the technical debt is not there anymore.

• But is this what you really want?
Potential vs. actual debt

• Potential debt
  – Looking at what you’ve done so far
  – Type 1: OK to detect with tools
    • see I. Gat & co. approach
  – Type 2: structural, architectural, or technological gap: Much harder to detect and evaluate
• Actual debt
  – When you know the way forward

Past? or future? Or both?

• Technical debt is not a mere function of the past (what you have done so far to reach the current state)
• It is also a function of what you want to do in the future
• So the cost cannot be assessed solely based on the current state.
A Better Metaphor?

- Unhedged Call Option
  – Chris Matt (2010), and Steve Freeman (2014)


Let’s try yet another metaphor

- Tech debt is Not necessarily a bad thing....
- Technical debt as an investment
**TD and Real Options**

P₁: $S₀ \rightarrow S₁$

Market loves it
+ $4M$

Market hates it
+ $1M$

\[ NPV (P₁) = -2M + 0.5 \times 4M + 0.5 \times 1M = 0.5M \]

Source: K. Sullivan, 2010 at TD Workshop SEI 6/2-3

---

**Taking Technical Debt has increased system value.**

**TD and Real Options (2)**

P₂: $S₀ \rightarrow S₁$

Market loves it
- $1M$

Market hates it
+ $1M$

\[ NPV (P₂) = -1M + 0.5 \times 3M + 0.5 \times 1M = 1M \]

Source: K. Sullivan, 2010
TD and Real Options (3)

NPV \((P_3)\) = \(-1M + 0.67 \times 2.5M + 0.33 \times 1M = 1M\)

More realistically:
Debt + interest
High chances of success

NPV \((P_3)\) = \(-1M + 0.67 \times 2.5M + 0.33 \times 1M = 1M\)

More realistically:
Debt + interest
50% interest
TD and Real Options (4)

Not debt really, but options with different values...
Do we want to invest in architecture, in test, etc...

Source: K. Sullivan, 2010

We are agile, so we’re immune!

In some cases we are agile and therefore we run faster into technical debt
Agile mottos

• “Defer decision to the last responsible moment”
• “YAGNI” = You Ain’t Gonna Need It
  – But when you do, much later, it is technical debt
  – Technical debt often is the accumulation of too many YAGNI decisions
• “We’ll refactor this later”
• “Deliver value, early”
• Tension between Big Upfront Design and Emergence
• You’re still agile because you aren’t slowed down by Tech Debt, yet.

Practical steps

From tactical (and simple) to more strategic (and sophisticated)
• Tactical
  – Short-term actions – limited scope
  – Actual means: use tools, add process steps, make an immediate plan

• Strategic
  – Long-term plan – wider scope
  – Process, management, education
  – Drive some of the tactical actions above

Practical steps (1) - Awareness

• Organize a lunch-and-learn with your team to introduce the concept of technical debt. Illustrate it with examples from your own projects, if possible.
• Create a category “TechDebt” in your issue tracking system, distinct from defects or from new features. Point at the specific artifacts involved.
• Standardize on one single form of “Fix me” or “Fix me later” comment in the source code to mark places that should be revised and improved later. They will be easier to spot with a tool.
Practical steps (2) - Identification

• Acquire and deploy in your development environment a static code analyzer to detect code-level “code smells”. (Do not panic in front of the large number of positive warnings).
• After some “triage” feed them in the issue tracking system, in the tech debt category
• At each development cycle (iteration), reduce some of the technical debt by explicitly bringing some tech debt items into your iteration or sprint backlog.

Practical steps (3) - Evaluation

• For identified tech debt items, give not only estimates of the cost to “reimburse” them or refactor them (in staff effort), but also estimate of the cost to not reimburse them: how much it drags the progress now. At least describe qualitatively the impact on productivity or quality. This can be assisted by tools from your development environment, to look at code churn, and effort spent.
• Prioritize technical debt items to fix or refactor, by doing them first in the parts of your code that are the most actively modified, leaving aside or for later the parts that are never touched.
Practical Steps (4) Architectural debt

- Refine in your issue tracker the TechDebt category into 2 subcategories: simple, localized, *code-level debt*, and wide ranging, structural or *architectural debt*.
- Acquire and deploy a tool that will give you hints about structural issues in your code: dependency analysis.

Practical Steps (5) Architectural debt

- Organize small 1-hour brainstorming sessions around the question: “What design decision did we make in the past that we regret now because it is costing us much?” or “If we had to do it again, what should have we done?”
  - This is not a blame game, or a whining session; just identify high level structural issues, the key design decisions from the past that have turned to technical debt today.
Practical steps (6) – Process improvements

• For your major kinds of technical debt, identify the root cause – schedule pressure, process or lack of process, people availability or turn over, knowledge or lack of knowledge, tool or lack of tool, change of strategy or objectives– and plan specific actions to address these root causes, or mitigate their effect.

• Develop an approach for systematic regression testing, so that fixing technical debt items does not run you in the risk of breaking the code.
  – Counter the “It is not really broken, so I won’t fix it.”

• If you are actively managing risks, consider bringing some major tech debt items in your list of risks.

So Technical debt...

• ... it’s messy

• Cannot isolate or tokenize
  – Lots of dependencies

• Cannot assess easily
  – Cost and value dependent on future evolution

• Polymorphic
  – Good & bad, costly and beneficial, harmful and innocuous
Key takeaways

• All software systems accumulate technical debt, which is different than defects.
• How much technical debt you suffer from depends on the future evolution of the system, not just its past.
• While code-level debt is easier to identify and remediate, architectural debt has the highest cost of ownership.
• Being agile does not make you immune to technical debt.

Reading on Technical debt

Managing Technical Debt
Reducing Friction in Software Development
Philippe Kruchten
Robert Nord
ipek Ozkaya

Also e-book
EPUB, MOBI, and PDF from
Informit.com
Slides will be at Philippe.Kruchten.com/Talks