Architectural technical debt: the hard bits

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
- Architectural technical debt
  - Form, symptoms
  - Causes & Consequences
- 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.

Technical Debt – Quick recap

- 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

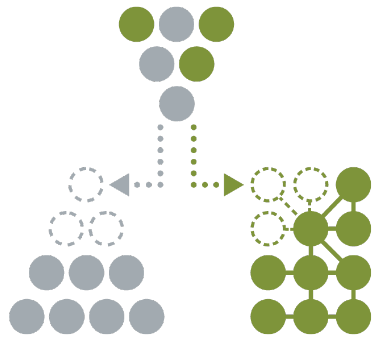
- Ward Cunningham, at OOPSLA 1992

"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 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.



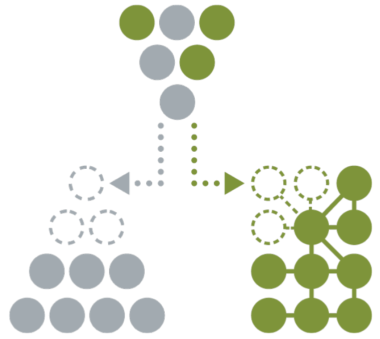
....

“Managing Technical Debt in Software Engineering,” Dagstuhl Reports, Vol. 6, Issue 4. http://www.dagstuhl.de/16162.

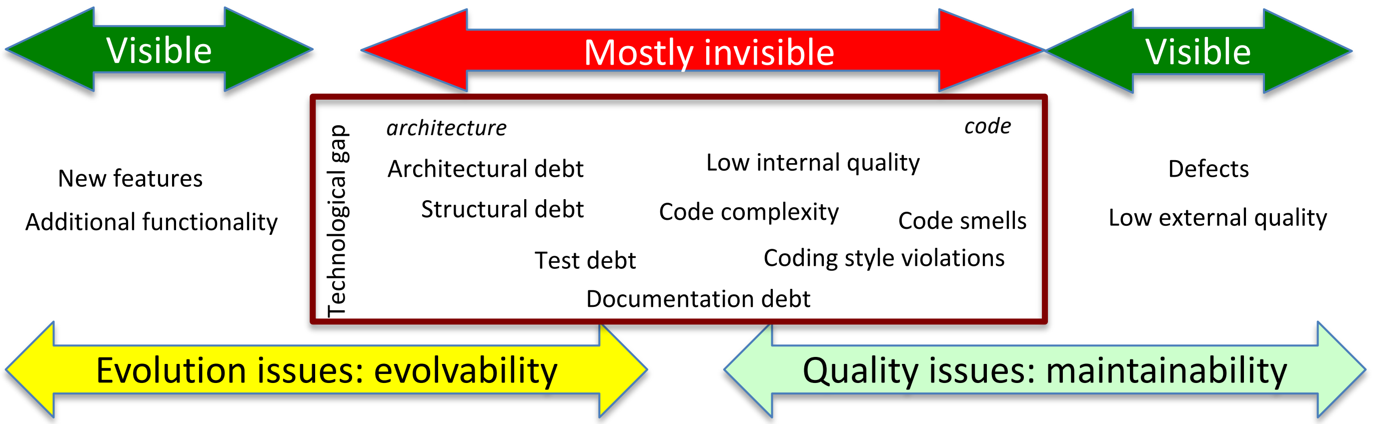
Technical Debt Definition (cont.)

….

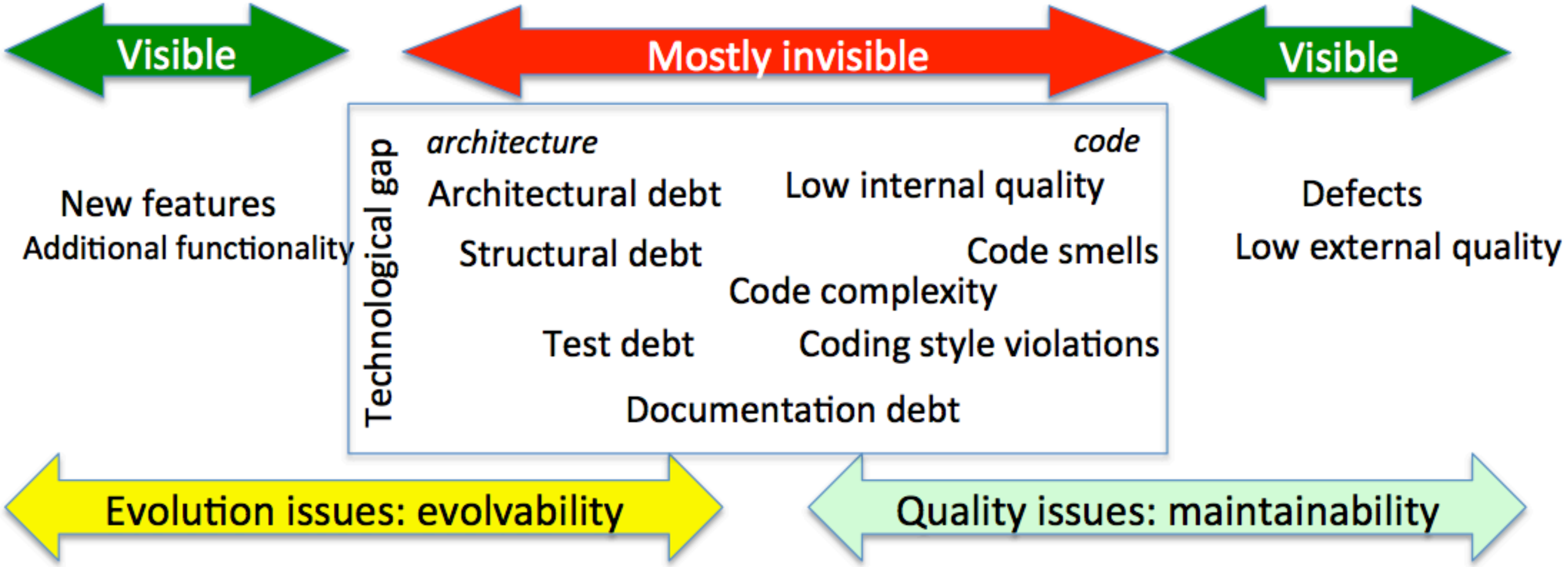
Technical debt presents an actual or contingent liability that impacts internal system qualities, primarily maintainability and evolvability.



The Technical debt landscape

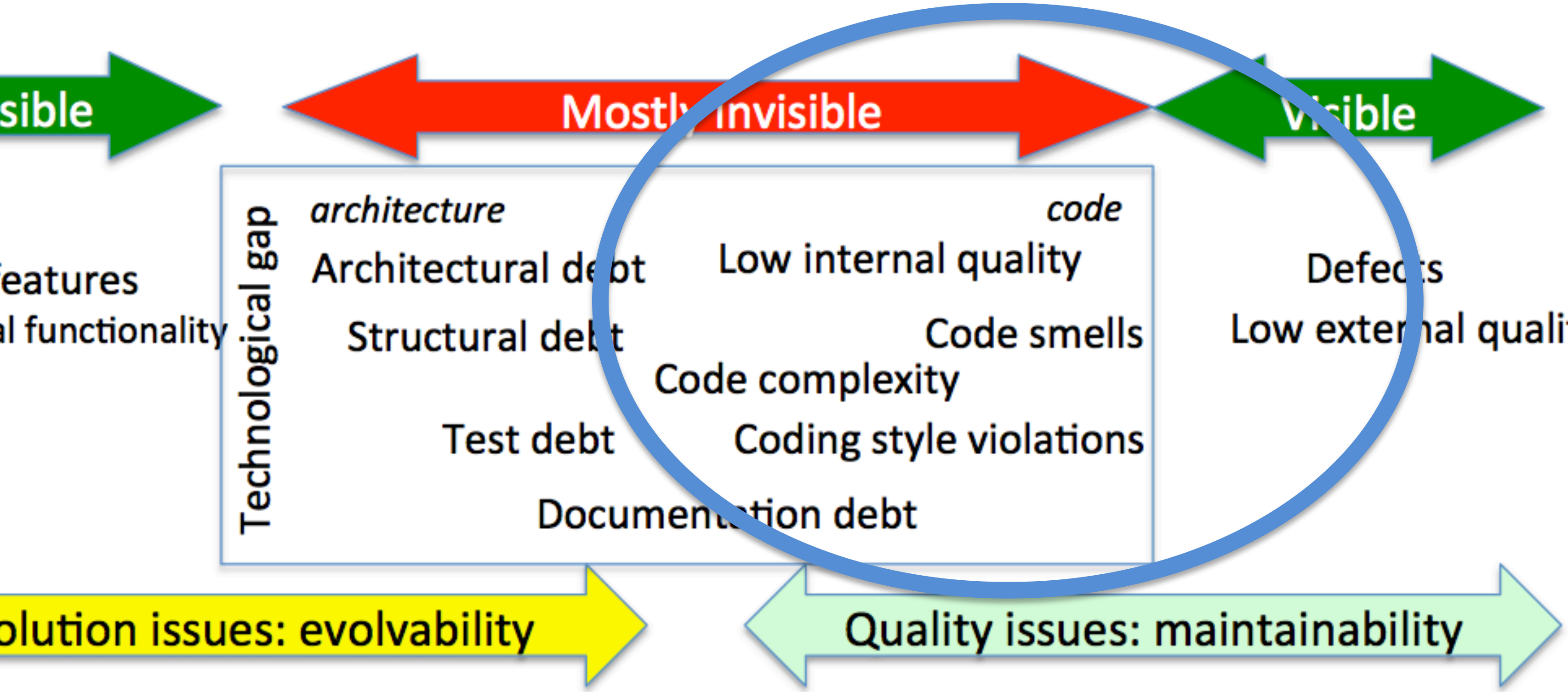


Zürich, May 2012


Visible
Mostly invisible
Visible
New features
Additional functionality
Technological gap
architecture
code
Architectural debt
Low internal quality
Structural debt
Code smells
Code complexity
Test debt
Coding style violations
Documentation debt
Defects
Low external quality
Evolution issues: evolvability
Quality issues: maintainability

TD in your backlog: negative value, invisible

	Visible	Invisible
Positive Value	New features Added functionality	Architectural, Structural features
Negative Value	Defects	Technical Debt

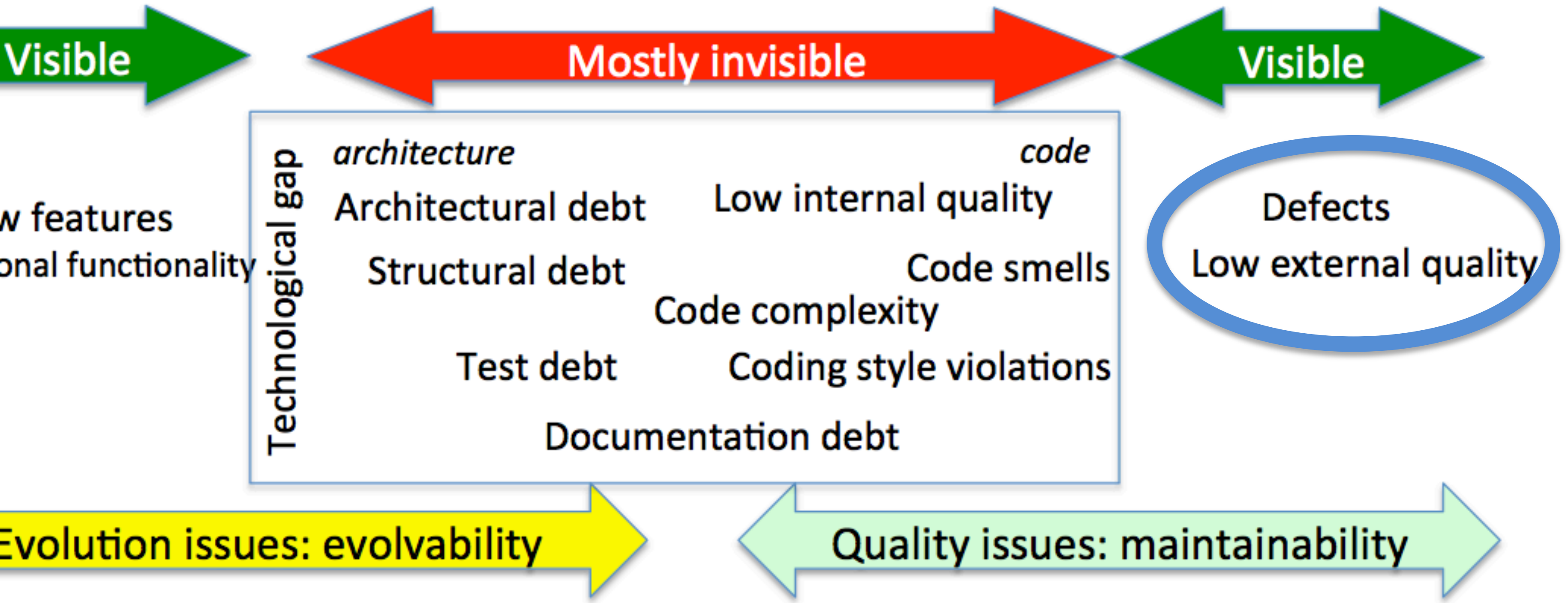

sible
Mostly invisible
Visible
architecture
code
Technological gap
eatures
l functionality
Architectural debt
Structural debt
Test debt
Documentation debt
Low internal quality
Code smells
Code complexity
Coding style violations
Defects
Low external quality
olution issues: evolvability
Quality issues: maintainability

Code-level technical debt

- Code smells
- Detected by static analysers
- Self-admitted technical debt

Common causes of technical debt

- Schedule pressure
- More schedule pressure
- Ignorance
- Success
- Environment evolution
  – Technical and business
- Sloppiness


Visible
Mostly invisible
Visible
w features
onal functionality
Technological gap
architecture
code
Architectural debt
Low internal quality
Structural debt
Code smells
Code complexity
Test debt
Coding style violations
Documentation debt
Defects
Low external quality
Evolution issues: evolvability
Quality issues: maintainability

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 !

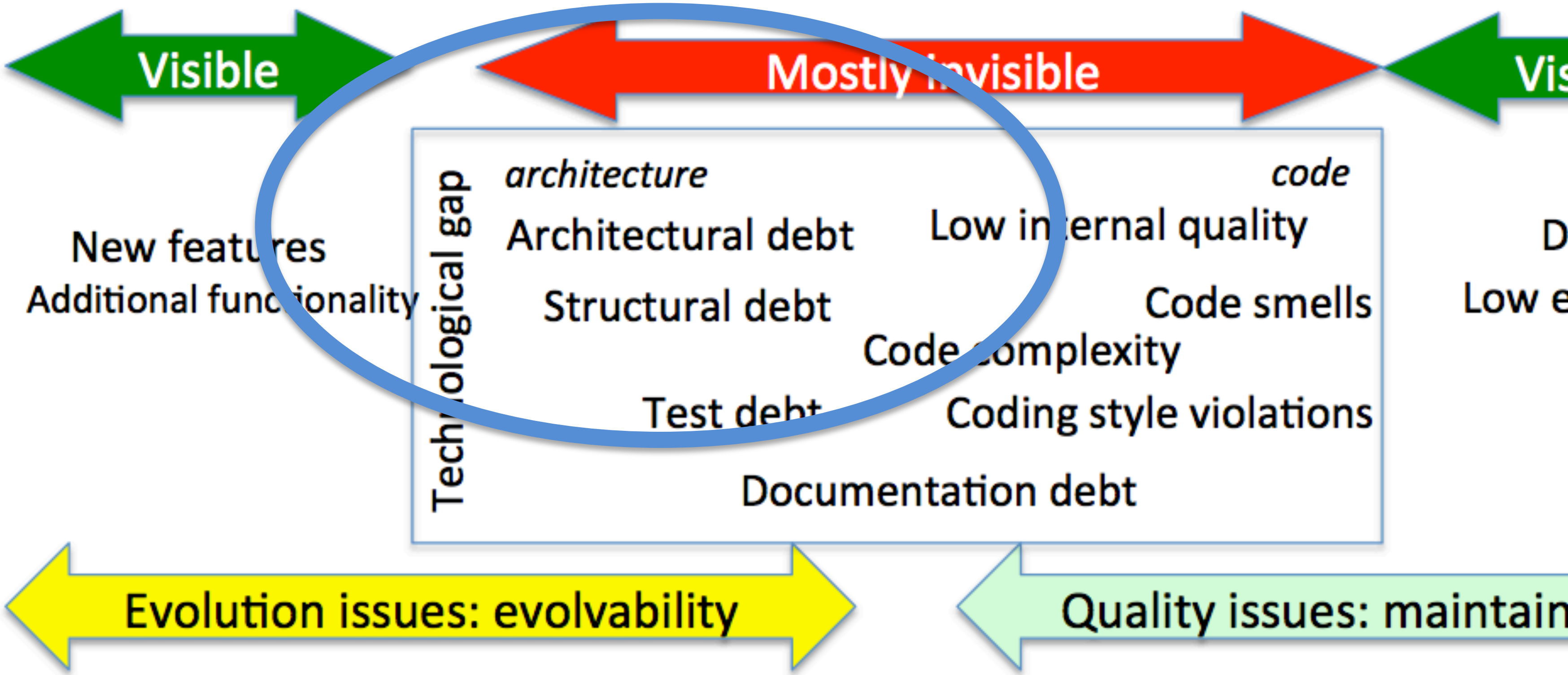
- *Sometimes, but rarely, the boundary is uncertain.*

Contrast…

Defect	Technical debt
• Visible	• Invisible
• External quality	• Internal quality
• Function of the past only	• Function of past and future
• Not a good investment	• Can be an investment
• Multiple possible causes	• Mainly triggered by schedule pressure
• All systems	• Large & long lived system

Strategy: 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


Visible
Mostly invisible
Vis
New features
Additional functionality
Technological gap
architecture
code
Architectural debt
Low internal quality
Structural debt
Code smells
Code complexity
Test debt
Coding style violations
Documentation debt
D
Low e
Evolution issues: evolvability
Quality issues: maintain

Study on Architectural Technical Debt

ELSEVIER

Journal of Systems and Software

Volume 176, June 2021, 110925

Building and evaluating a theory of architectural technical debt in software-intensive systems ☆

Roberto Verdecchia [a], Philippe Kruchten [b], Patricia Lago [a, c], Ivano Malavolta [a]

https://doi.org/10.1016/j.jss.2021.110925 (open access)

Kinds of Architectural Debt

- Three main buckets:

1. Bad architectural design choices
2. Good choices, but wrong context
3. Good choices, bad implementation

Examples of 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

R. Verdecchia 2020

Causes of architectural debt

External causes *vs.* internal causes

- Time pressure (9/10)
- Lack of architectural knowledge
- Overly complex product development process
- Human factors: bias, lack of experience, etc.
- Lack of anticipation
- Lack of architectural documentation

- The passing of time…

R. Verdecchia 2020

Consequences (and often symptoms)

- Carrying costs, reduced development velocity
- Loss of business opportunities
- Loss of external quality:
  - Increased defects
  - Inability to scale
- Dependence on specific staff

Bring me tools!

- Static analyzers will detect much of code-level technical debt.
- More significant technical debt items (structural, 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.

Potential vs. actual debt

- Potential debt
  - Looking at what you've done so far
  - Code level: static analyzers
  - Structural, architectural, or technological gap: Much harder to detect and evaluate
- Actual debt
  - When you know the way forward

K.Schmid 2013

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.

It’s just a metaphor!

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

Managing Architectural Tech Debt

- Live with it   (half of the cases)

- Minor refactorings (limited benefits)
- Major refactorings (costly, product delays)
- Re-design and reimplementation (costly, very risky)

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

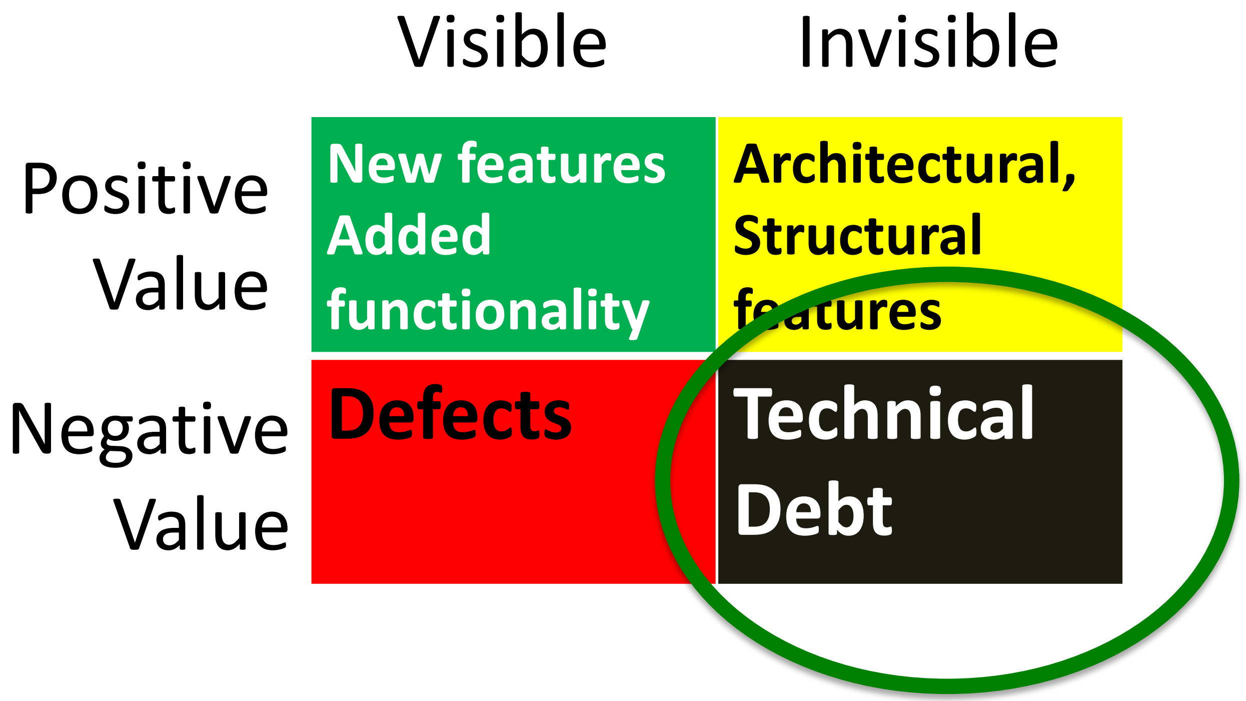
Practical steps (1) - Awareness

- 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.

TD in your backlog: negative value, invisible



	Visible	Invisible
Positive Value	New features Added functionality	Architectural, Structural features
Negative Value	Defects	Technical Debt

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; especially architectural debt.

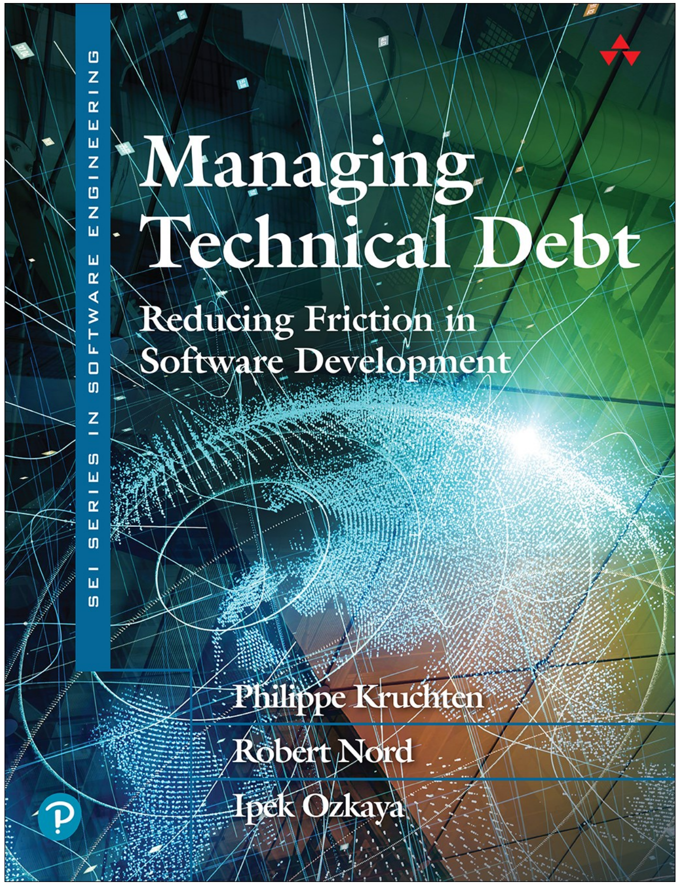
- Cannot isolate or tokenize
  - Lots of dependencies, little tool support
- Difficult to assess
  - Cost and value dependent on future evolution
- Polymorphic
  - Good & bad, costly and beneficial, harmful and innocuous

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.

Reading on Technical debt

June 2019
Addison-Wesley Professional
Boston
272 p.
978-0135645932


SEI SERIES IN SOFTWARE ENGINEERING
Managing Technical Debt
Reducing Friction in Software Development
Philippe Kruchten
Robert Nord
Ipek Ozkaya

Chapter 6 is about architectural debt

Also e-book
EPUB, MOBI, and PDF from
Informit.com

Study on Architectural Technical Debt

R. Verdecchia, Ph. Kruchten, P. Lago, I. Malavolta:

“Building and evaluating a **theory of architectural technical debt** in software-intensive systems,”

Journal of Systems and Software,

Volume 176, June 2021.

online February 27, 2021

https://doi.org/10.1016/j.jss.2021.110925 (open access)

More pointers…

- Dagstuhl report: http://www.dagstuhl.de/16162
- What color is your backlog? https://tinyurl.com/y6f7vhpx
- Concrete things you can do about your tech debt: https://philippe.kruchten.com/2017/02/14/concrete-things-you-can-do-about-your-technical-debt/

- A. Martini et al., *Investigating architectural technical debt accumulation*… https://doi.org/10.1016/j.infsof.2015.07.005
- And a couple of earlier papers

Slides will be at Philippe.Kruchten.com/Talks