

Antonio Martini

Professor at University of Oslo

ITEVU4350

23-04-2026

MANAGING TECHNICAL DEBT



Antonio Martini



- ✗ Italian and Norwegian citizen
 - ✗ I don't drink Martini ☺
 - ✗ Lived 15 years in Scandinavia (Sweden, Norway)
 - Survived many winters!
- ✗ Professor at the University of Oslo
 - ✗ Group leader of the SE group
- ✗ Research
 - ✗ Technical Debt
 - ✗ Architecture
 - ✗ Agile
 - ✗ ... and of course AI4TD and AI engineering
- ✗ Previously
 - ✗ Software Developer
 - ✗ Independent Consultant (Ericsson, Volvo)
 - ✗ Principal Strategic Researcher, CA Technologies (Broadcom)
 - ✗ Founder and advisor for two startups



Agenda

- 1st part:
 - Motivation and definition of Technical Debt
 - Causes and dangers
- 2nd part
 - Managing Technical debt
 - Proactive, continuous, reactive approaches
- 3rd part
 - Assessing Technical Debt Management
 - Prioritizing technical debt
- Lunch 12.00-13.00
- 4th part
 - Group work
 - Work with Robin Börjesson, architect at DNB
- 5th part
 - Technical Debt and AI
 - A look at the crystal ball and discussion



Results from 15 years of collaboration with industry on managing Technical Debt



TechDebtOps – A 3-year project funded by the Research Council of Norway

**Data-driven continuous
management of technical debts
for sustainable software
development**

AKVAGROUP™

knowit

 **VISMA**

VOLVO



SAAB



ca
technologies

SIEMENS

Nordea



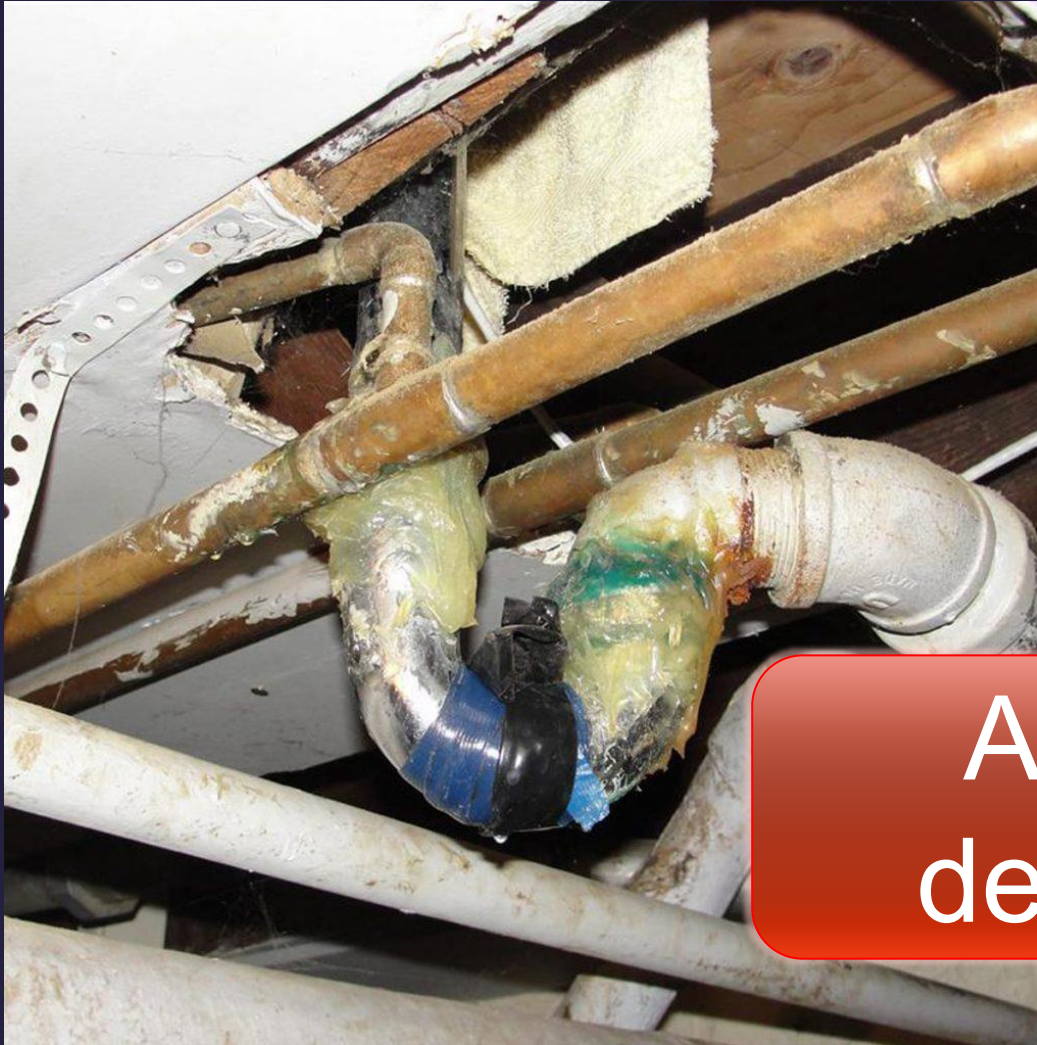
Technical Debt - definitions

What the users see



Wonderful
features!

What the developers see

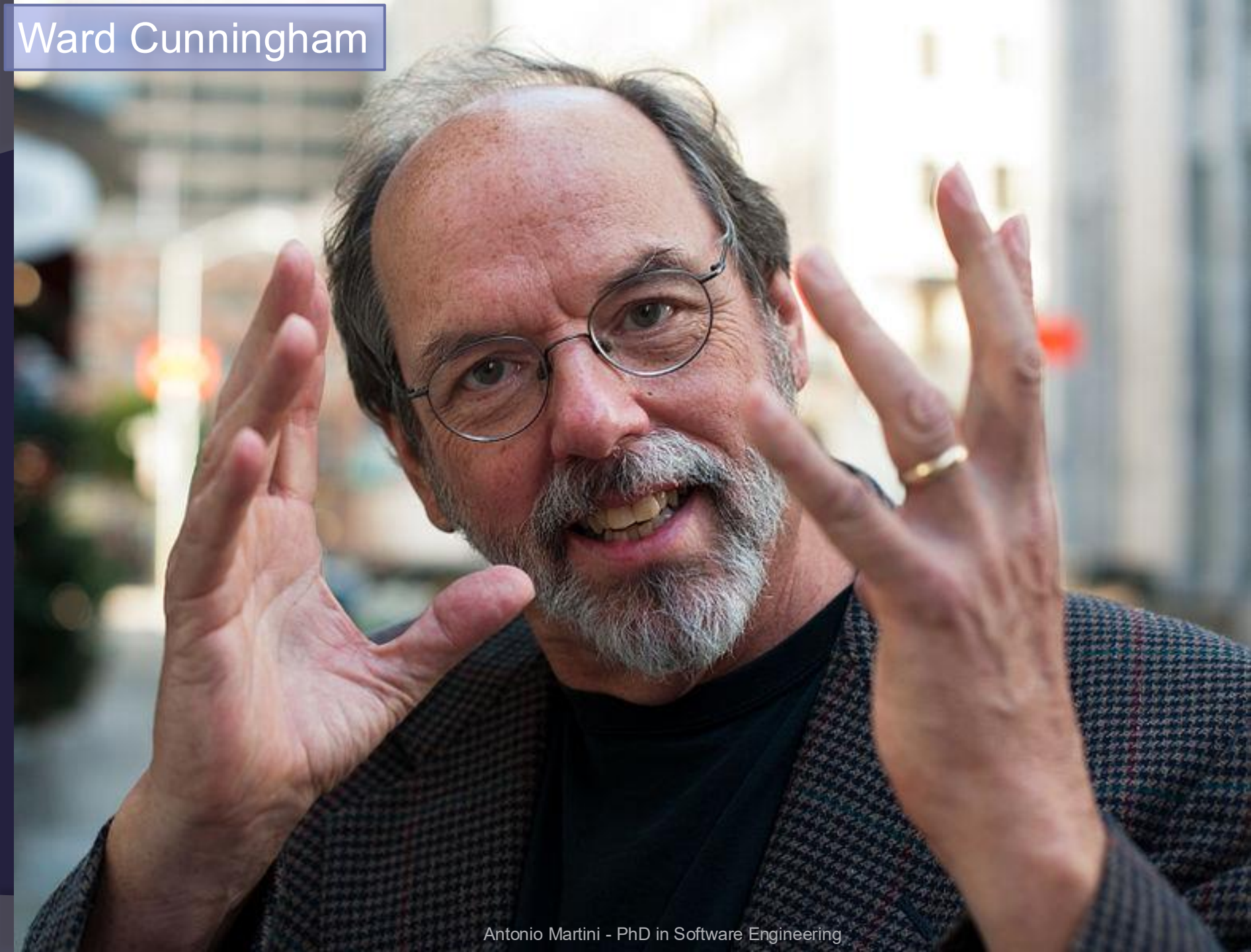


Awful
design!

Technical Debt in a nutshell



Ward Cunningham



"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..."

"Every minute spent on not-quite-right code counts as interest on that debt"

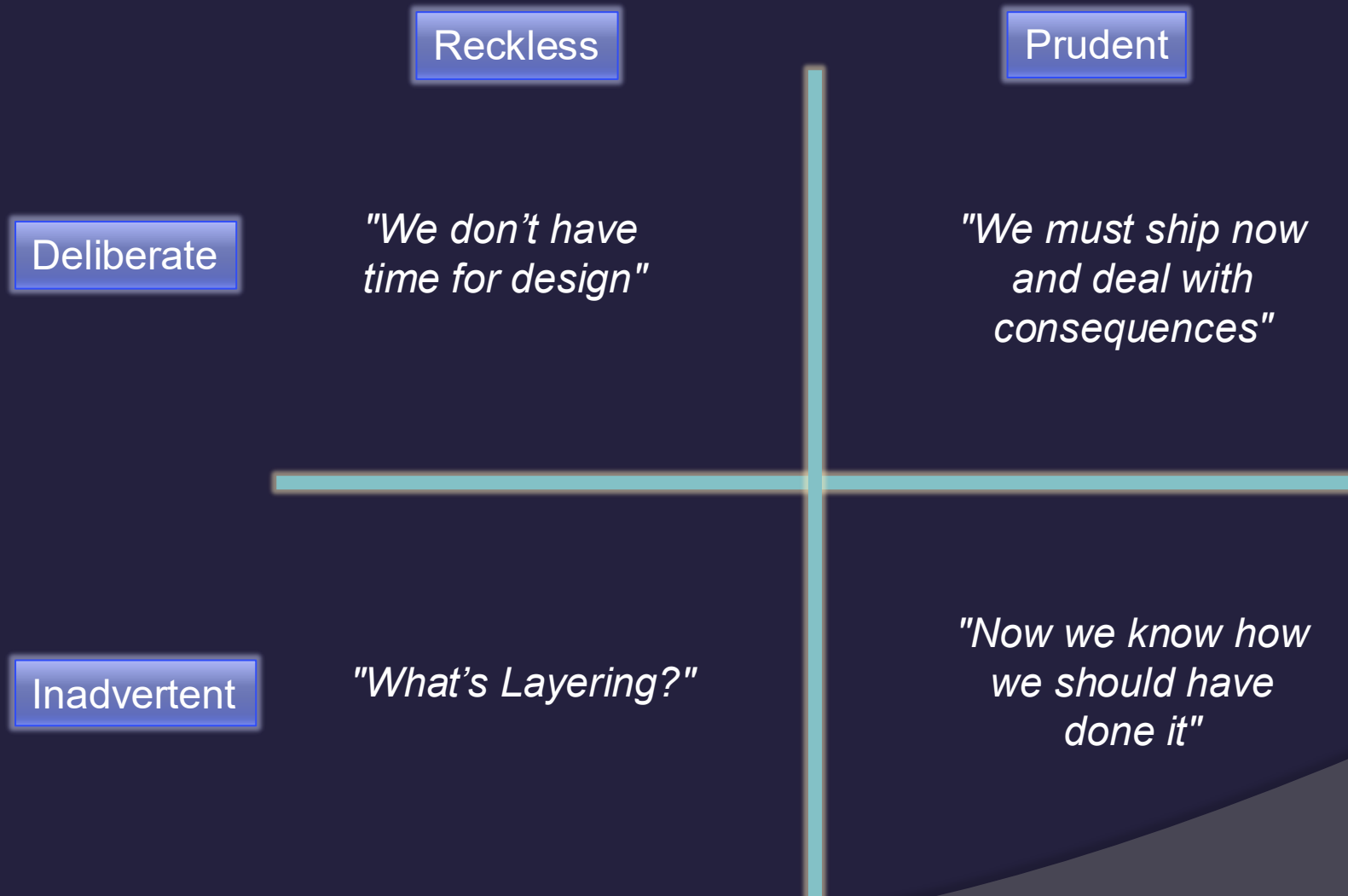
Definition of Technical Debt

- *"a technical or design solution that has **benefits** in the **short term**, but makes development more **costly or impossible** in the **future**"*
- *Complete definition in **

* P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)

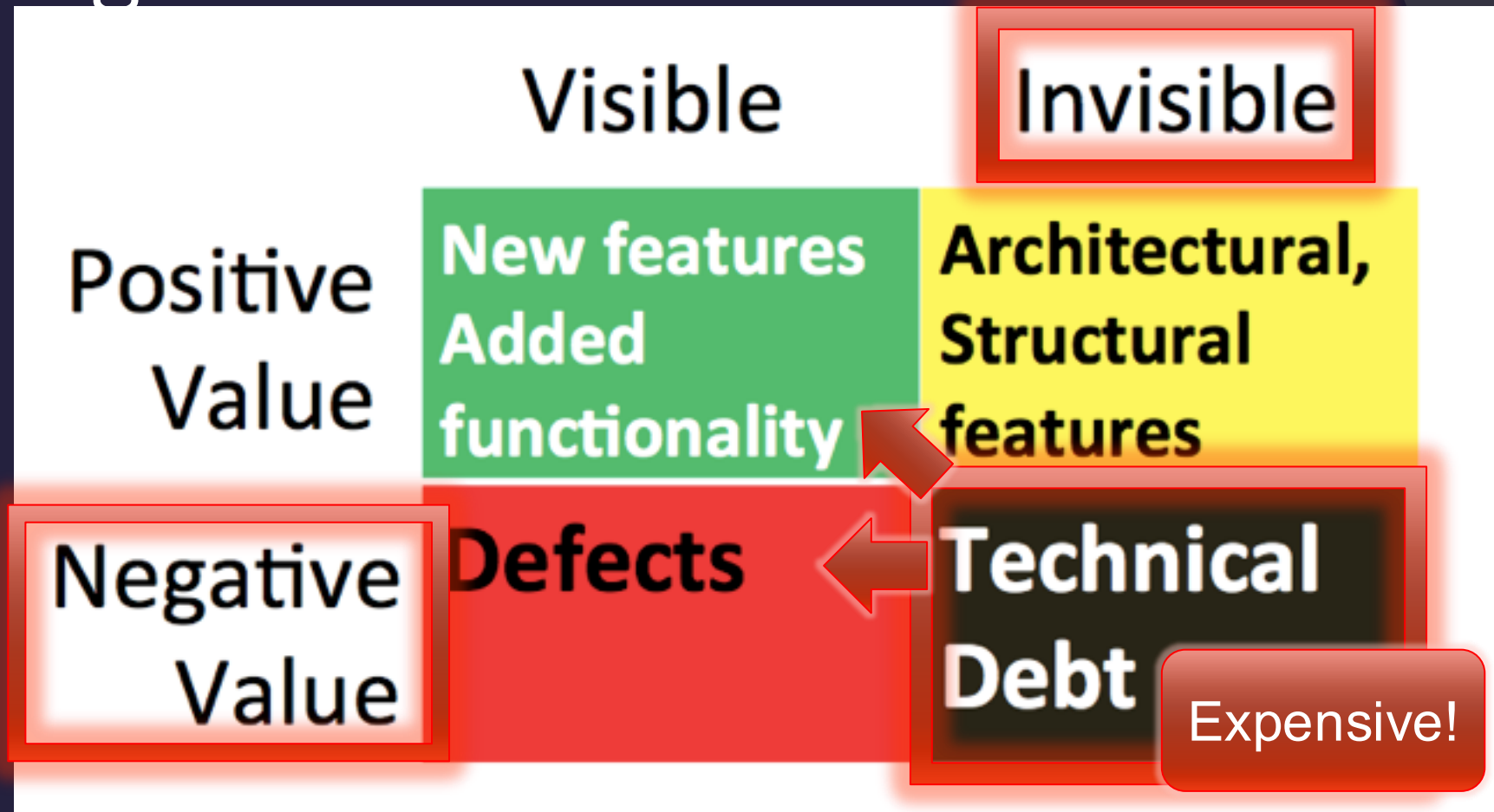


Attitude towards Technical Debt



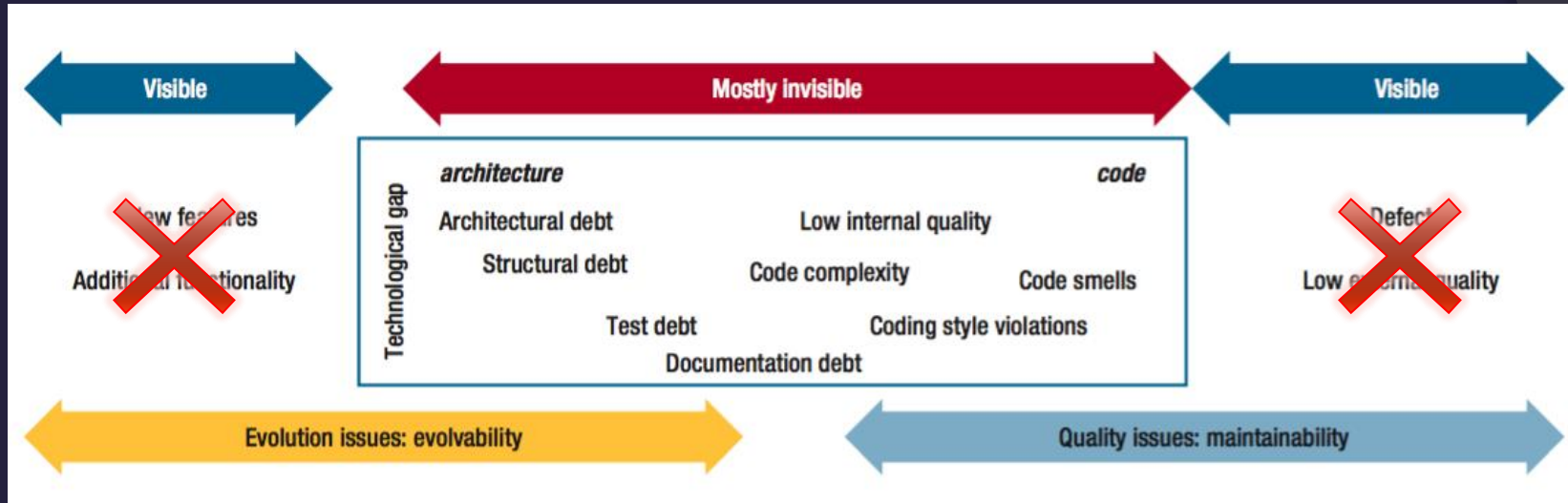
* martinfowler.com/bliki/TechnicalDebtQuadrant.html

Technical Debt – Invisible and Negative



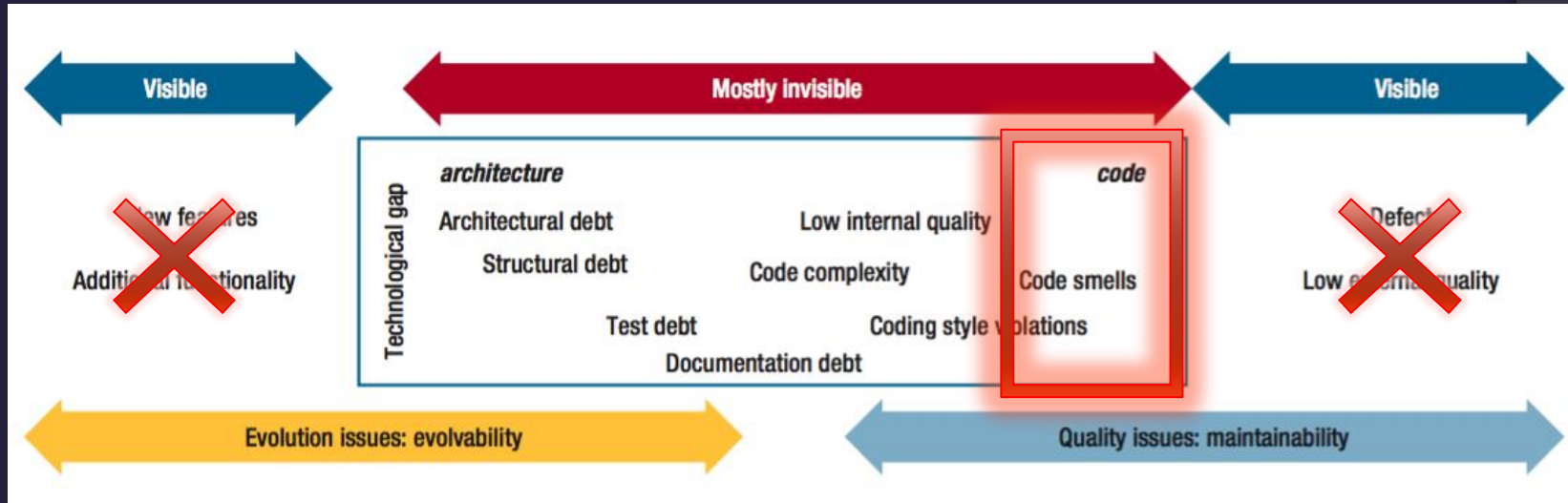
P. Kruchten, R. L. Nord, and I. Ozkaya, “Technical Debt: From Metaphor to Theory and Practice,” *IEEE Software*

The TD landscape



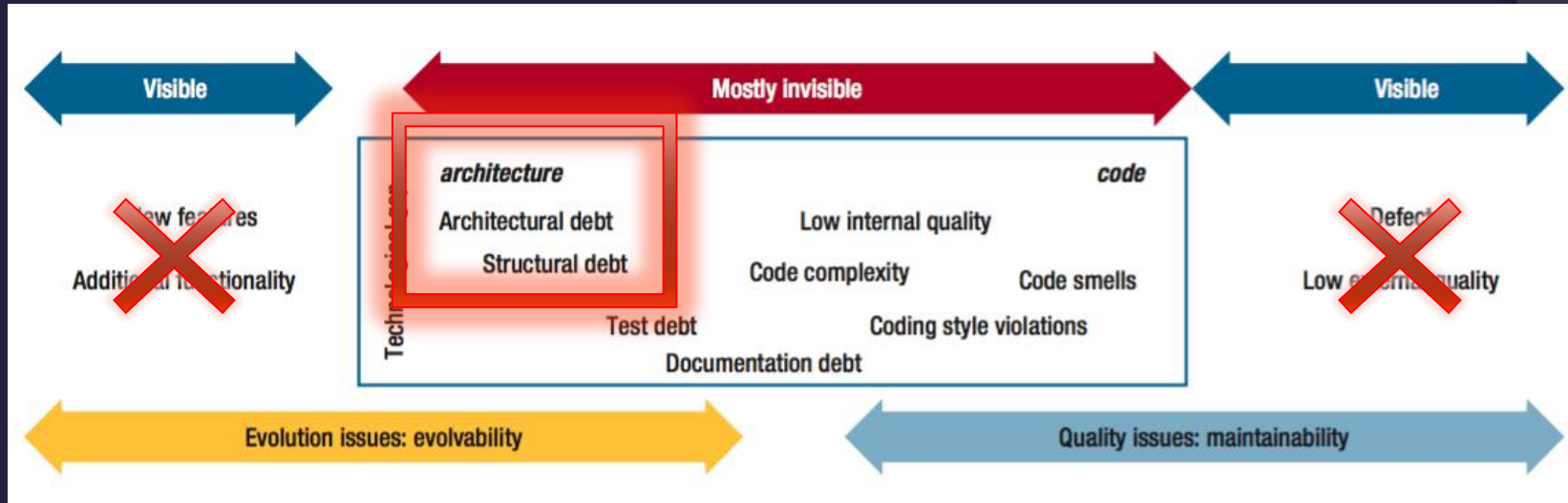
P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Software*

Example of Code Debt



P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Software*

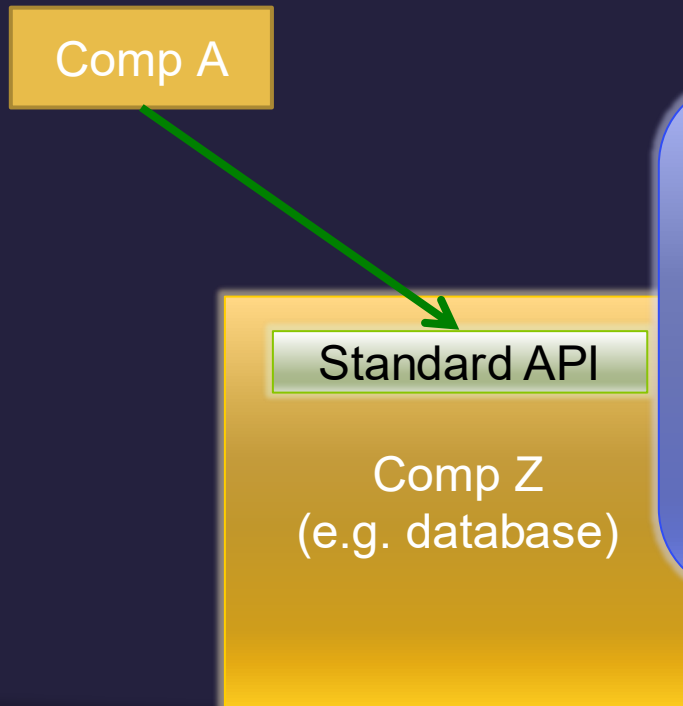
Example of Architectural Technical Debt



P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Software*

Optimal architectural decision

- Example:
 - Standard public API



Let's put a standard API here... so later we can update the component independently



During feature development...

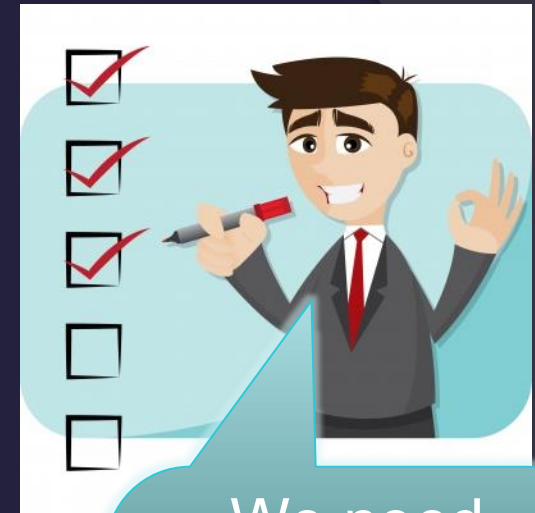
No problem, let's add a component B. The teams will use the standard API!

Comp A

Comp B

Standard API

Comp Z

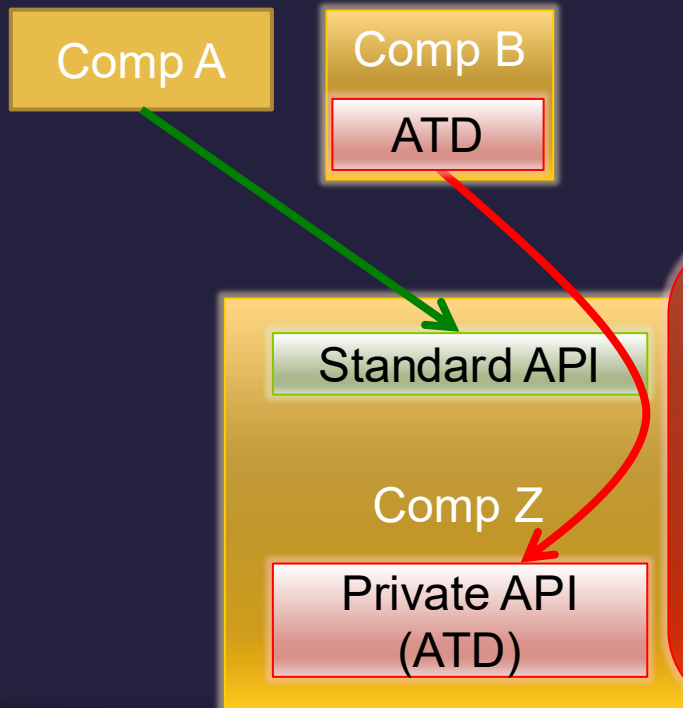


We need these new features! Our competitor is already delivering them!

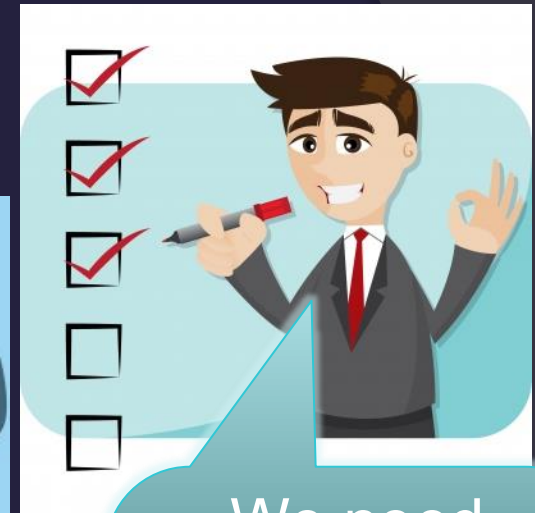


...with fast delivery comes...

● Deliver fast!



We have to deliver fast, let's use the private API... we'll change it later



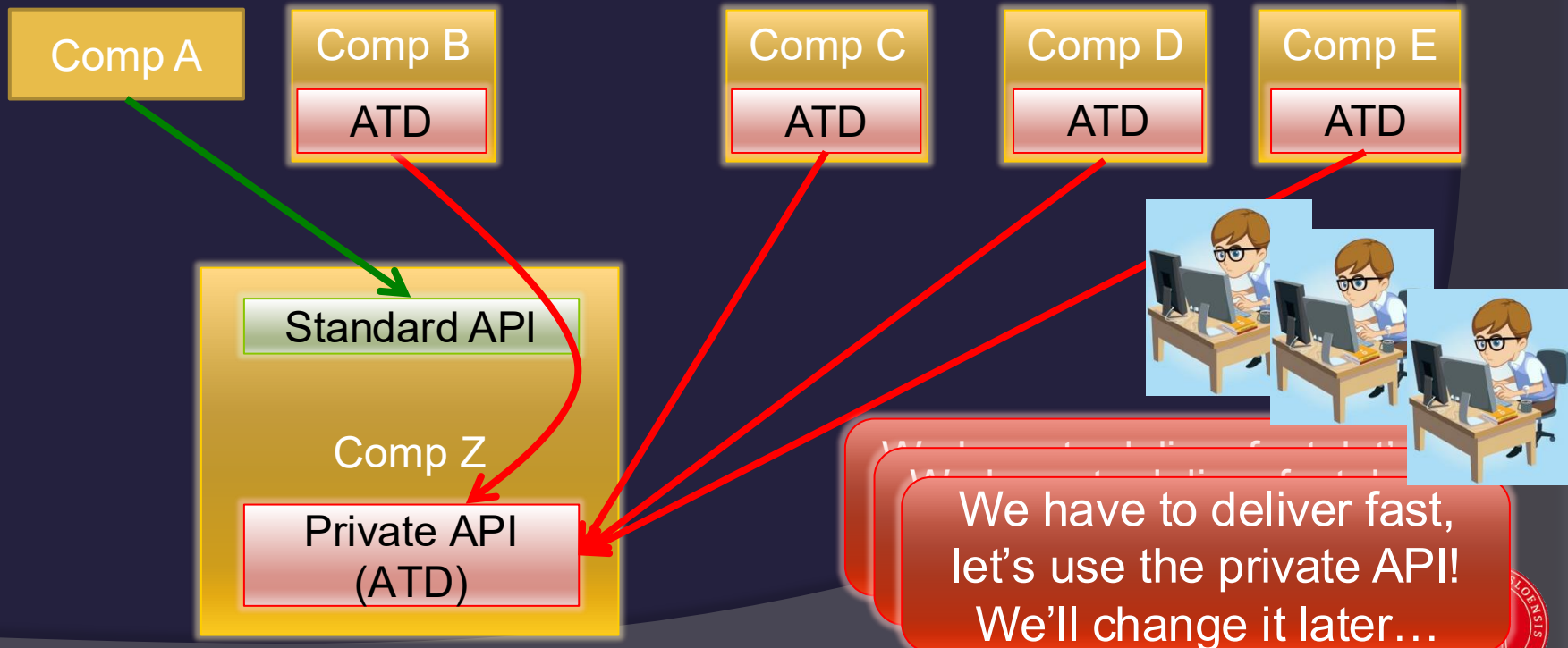
We need these new features! Our competitor is already delivering them!

Fast!

...the accumulation of sub-optimal decisions...

- The violation is spreading to many components

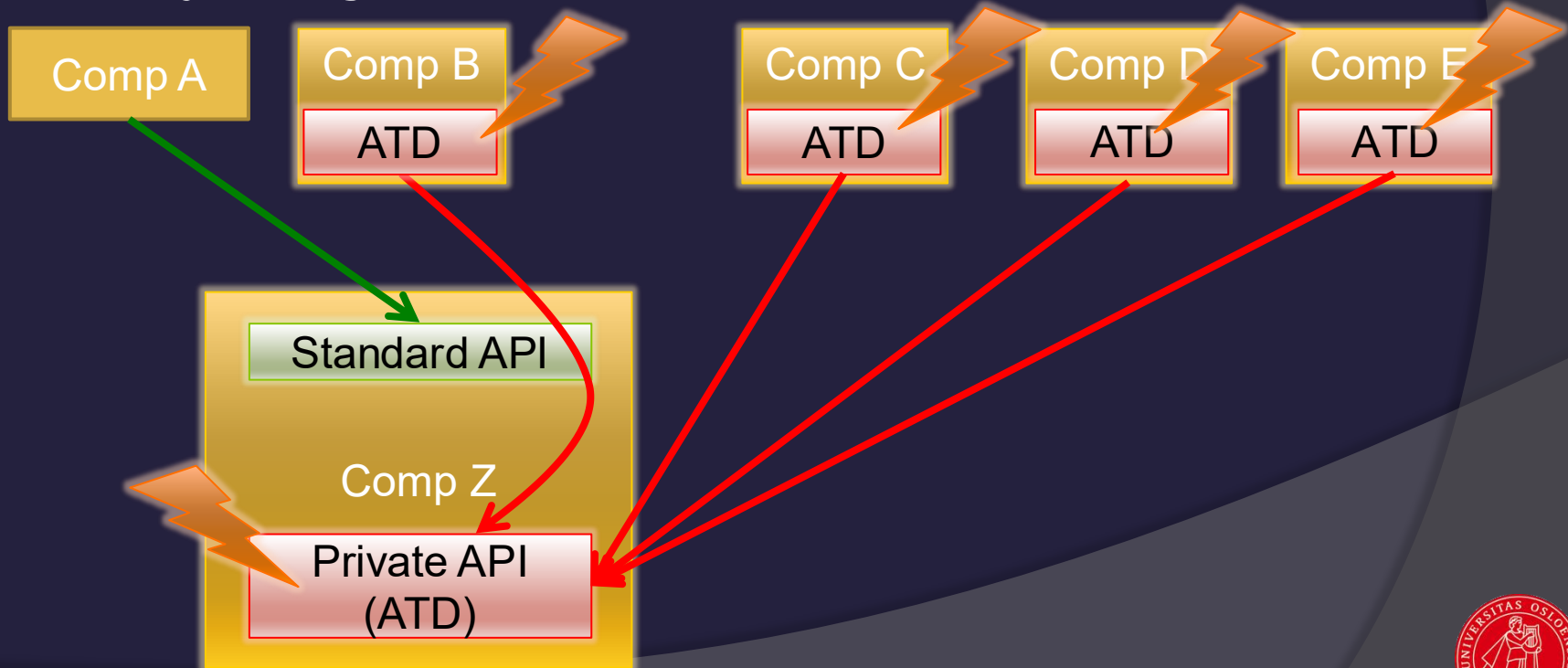
Fast!



...generate costly changes...

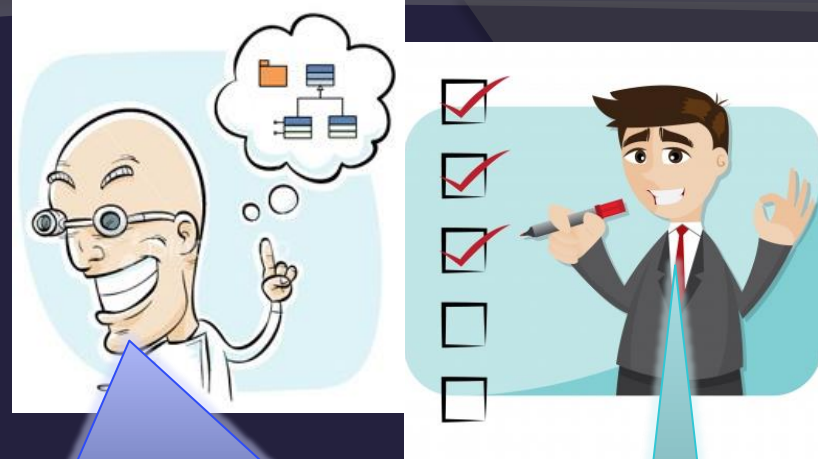
- Every time we change the private API we need to change everything else

Fast!

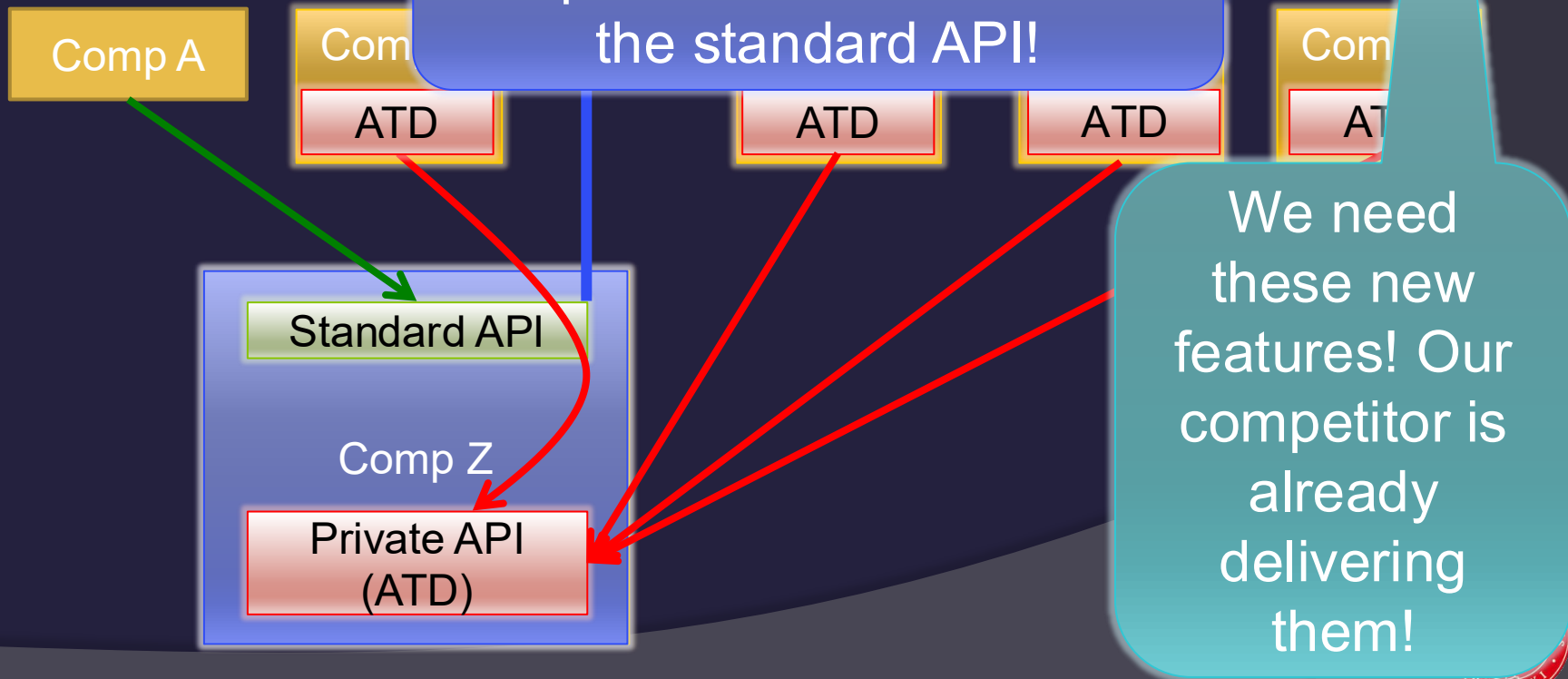


...until, one day...

● New requirement

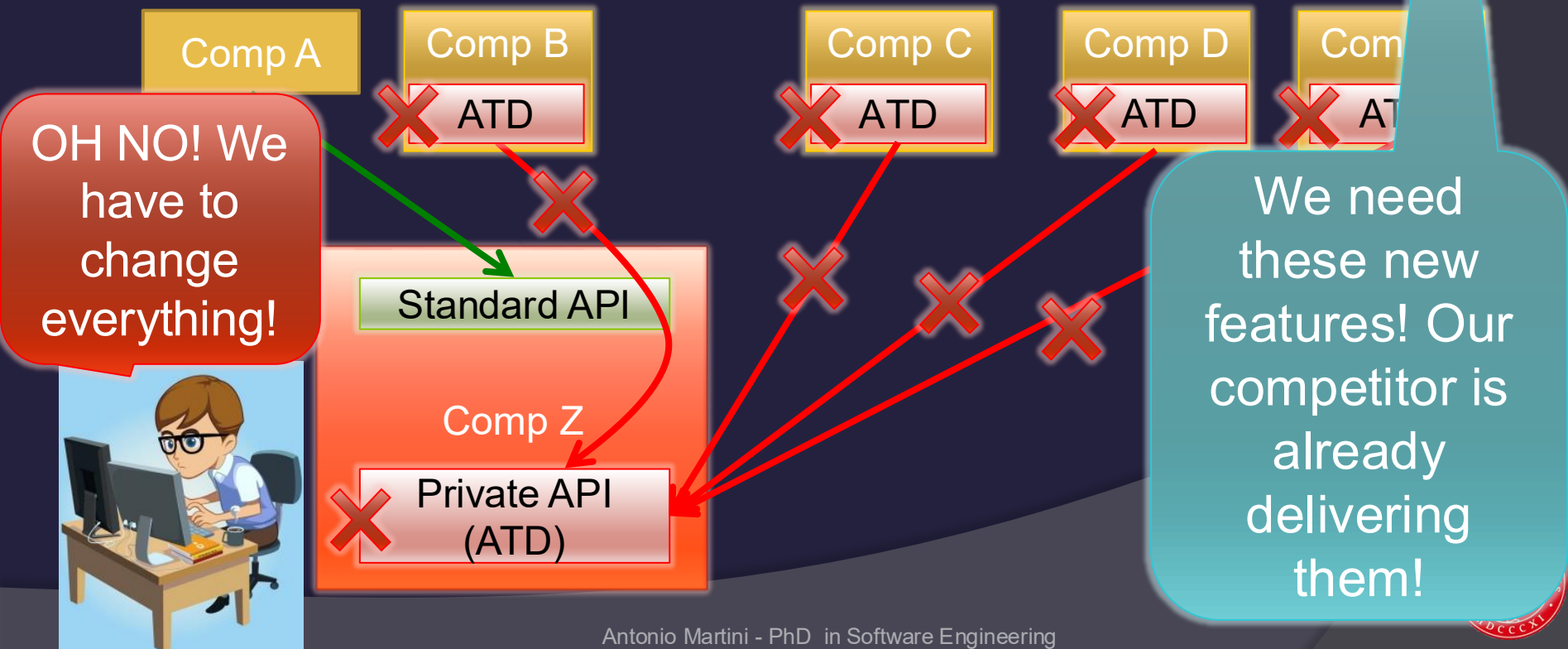


Ok, we can replace this component. The teams used the standard API!



...the development is not fast anymore...

- *Costly* to remove the violation and *difficult to estimate the impact*



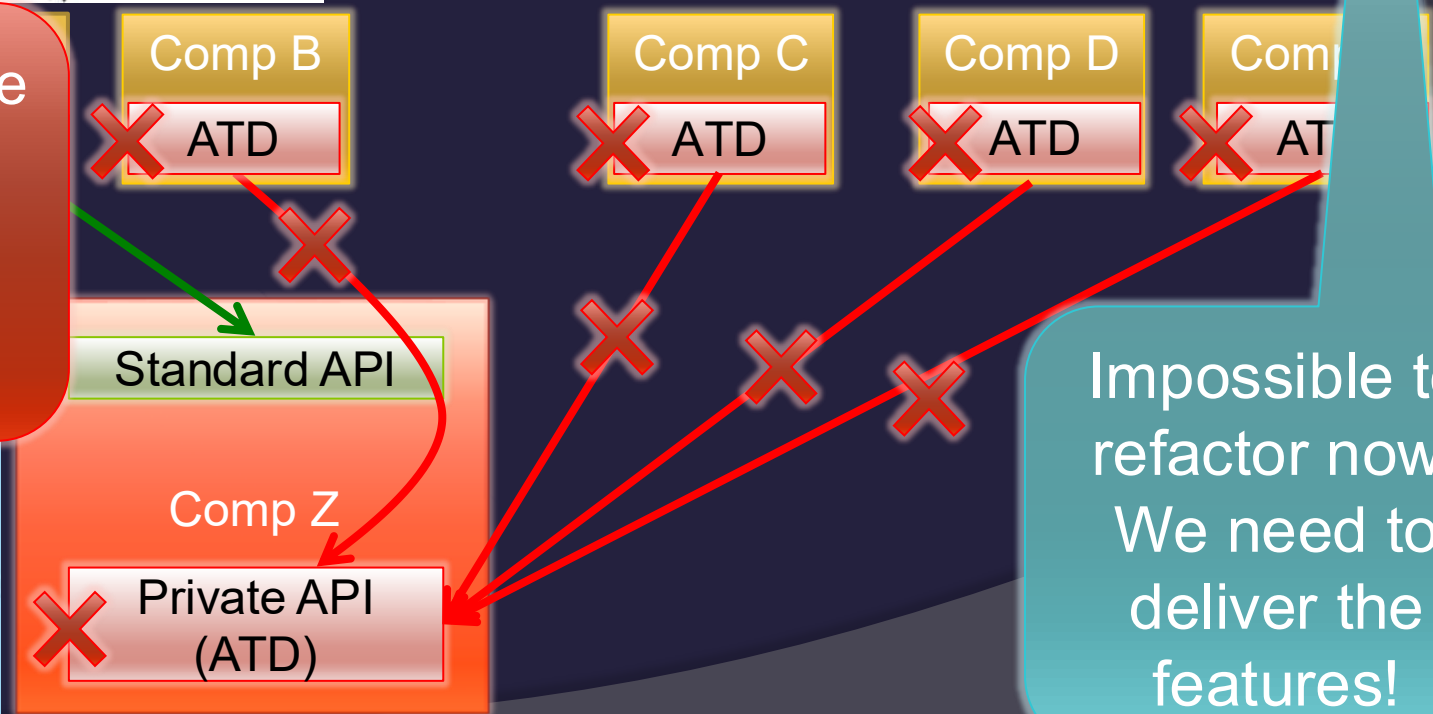
...and a crisis starts.



We have to refactor, but we need time...



So should we refactor or continuing with other features?



Impossible to refactor now!
We need to deliver the features!

So what is Architectural Technical Debt?

- ⦿ **Non-allowed** dependencies = Debt
- ⦿ **Cost of removing** dependencies = Principal
- ⦿ **Extra evolution cost**
Important

= Interest
- ⦿ Other **impacts**
 - E.g. increasing principal

The dangers of Technical Debt

So... what's the problem?



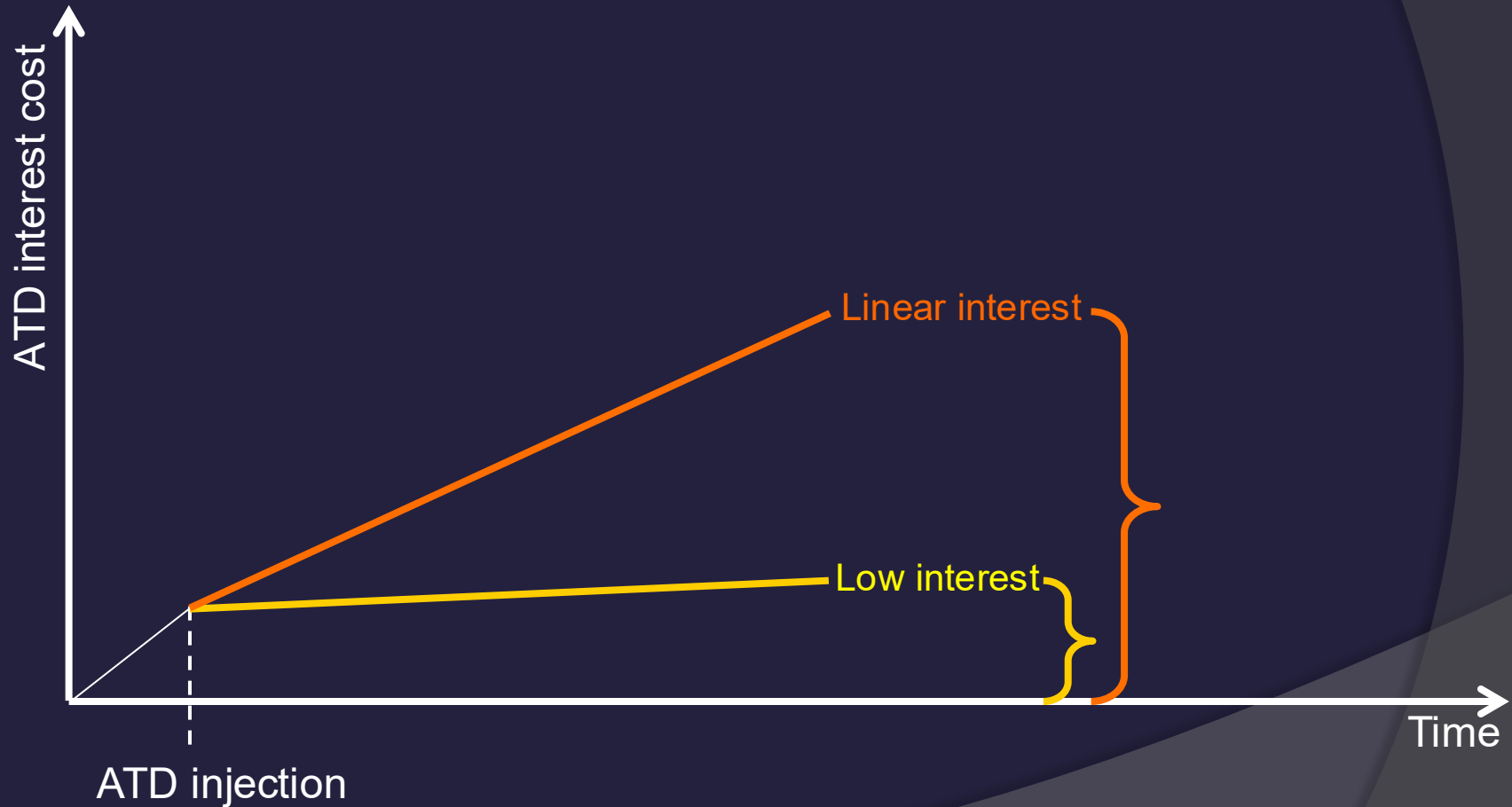
What's the problem? Growing interest



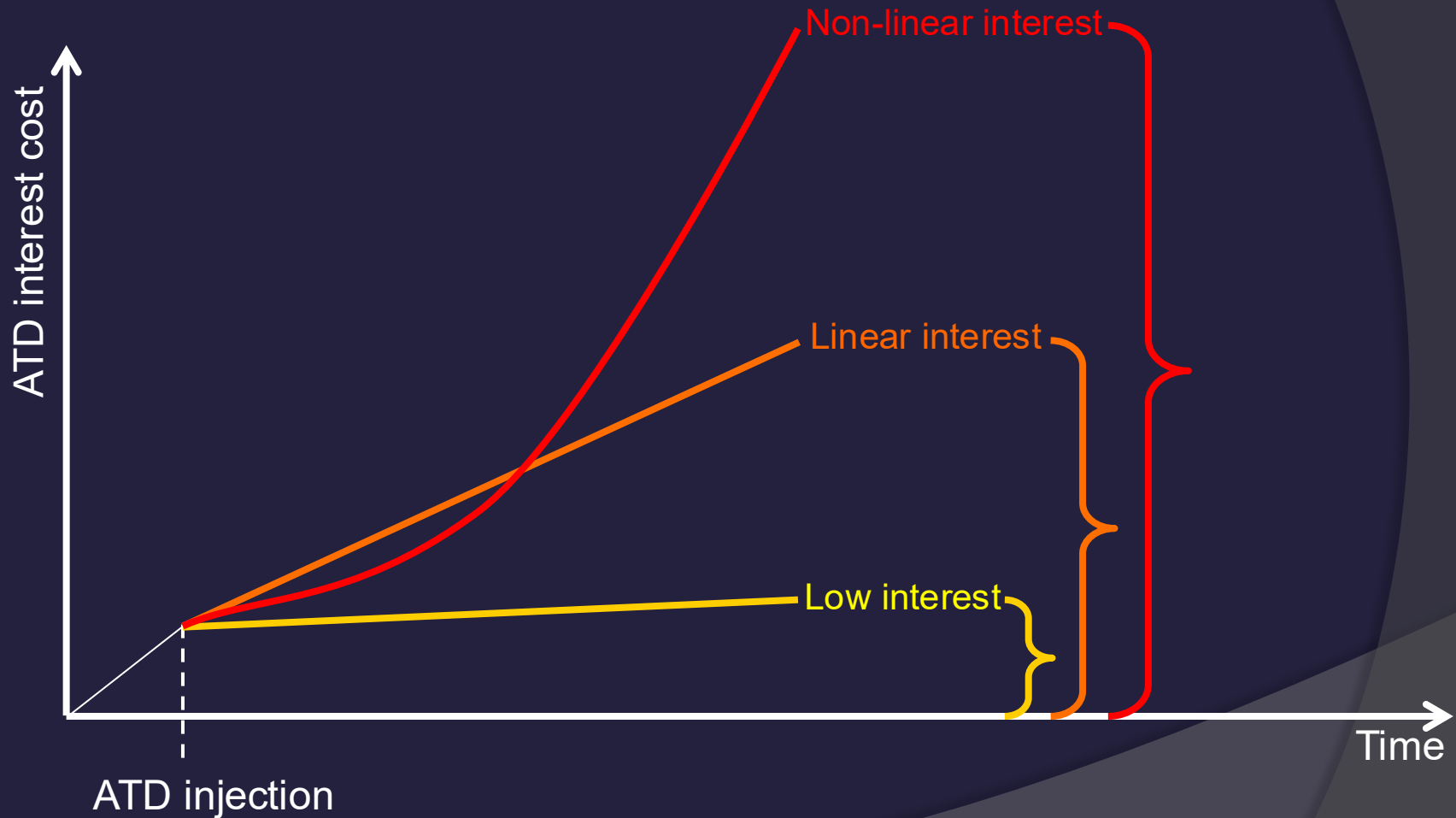
Growing interest



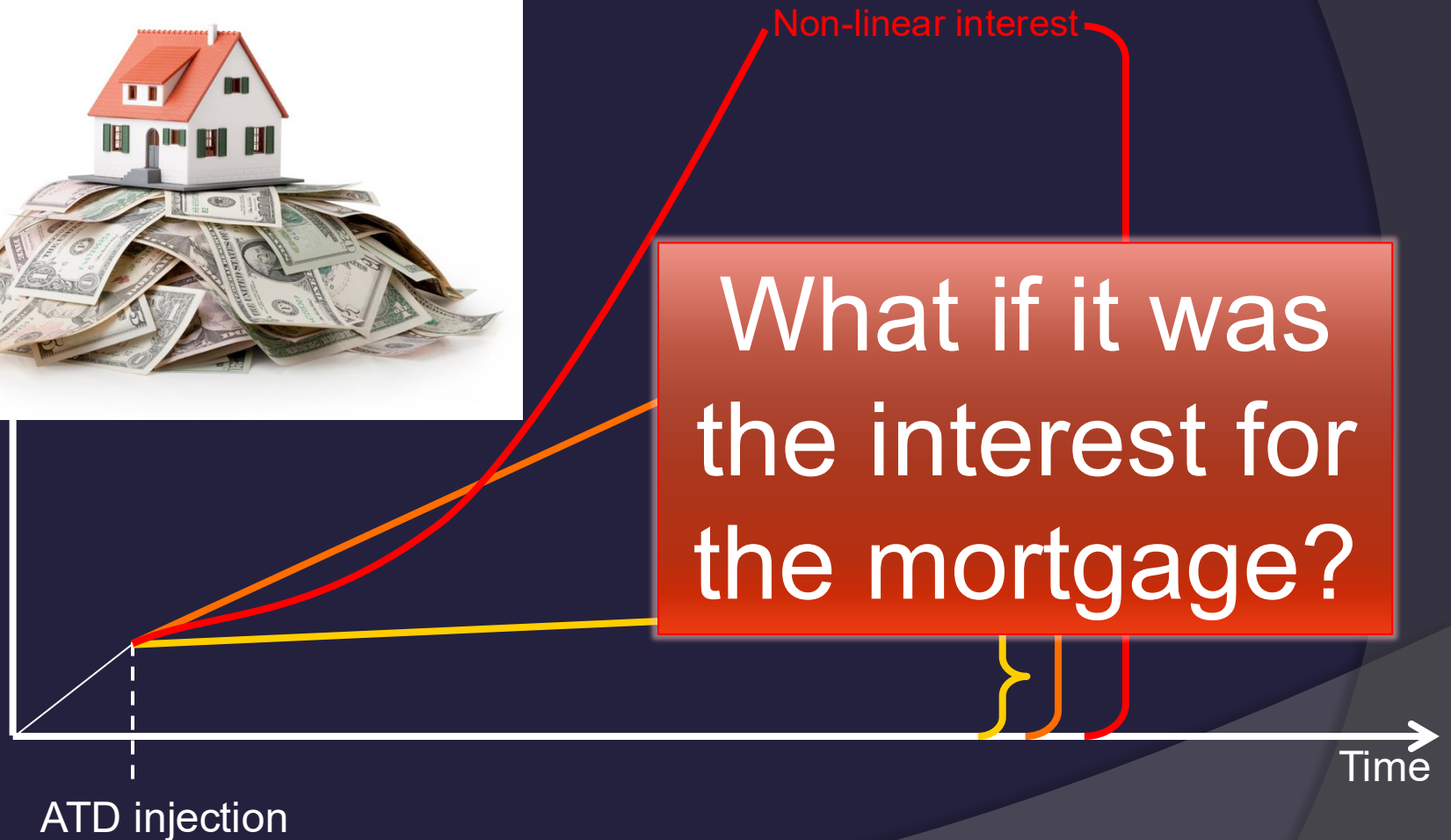
Growing interest



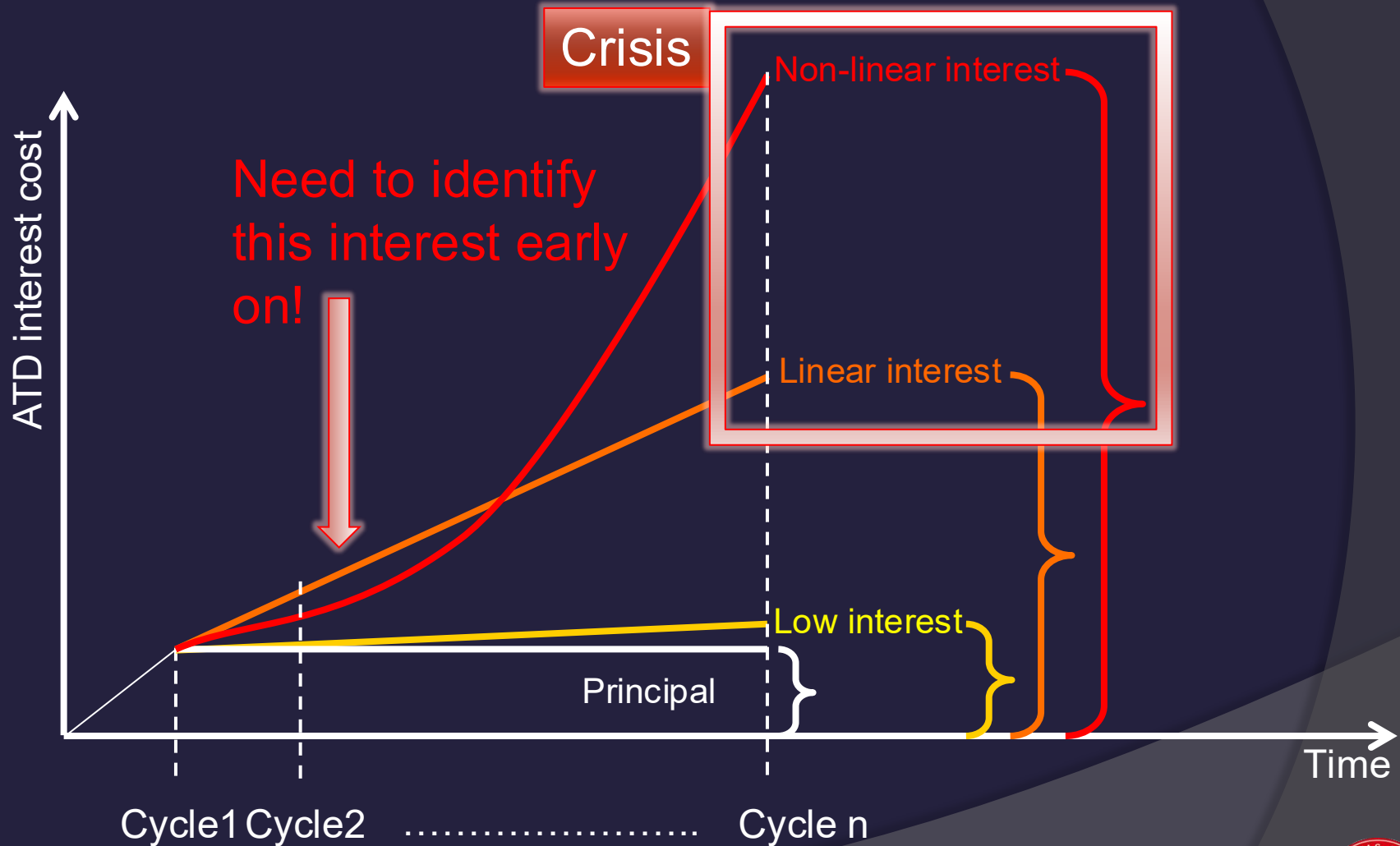
Growing interest



Growing interest



Growing interest



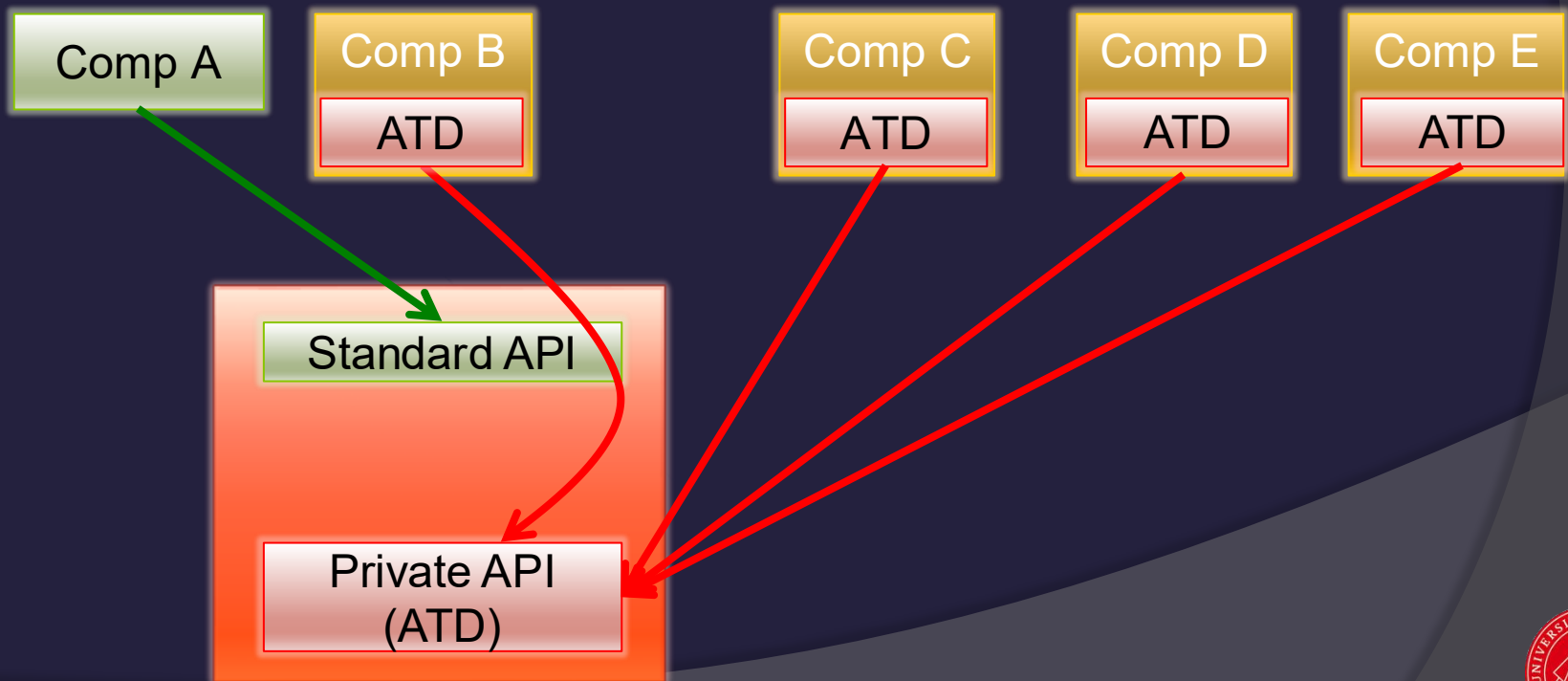
Martini, Bosch: "The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles,"
in *accepted for publication at WICSA 2015*, Montreal, Canada.

Antonio Martini - PhD in Software Engineering

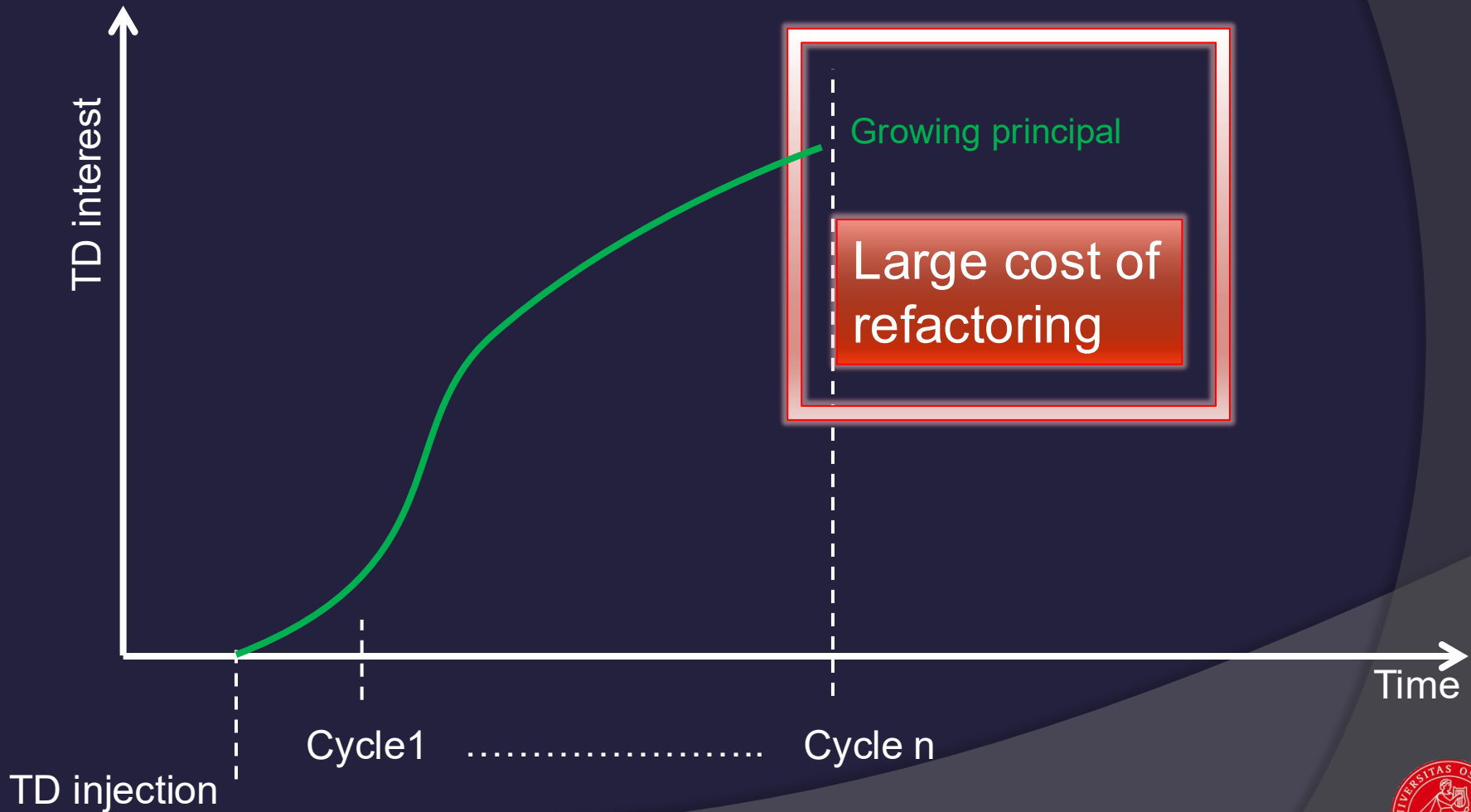


Growing interest

- *Example of TD and interest growing as the code grows*

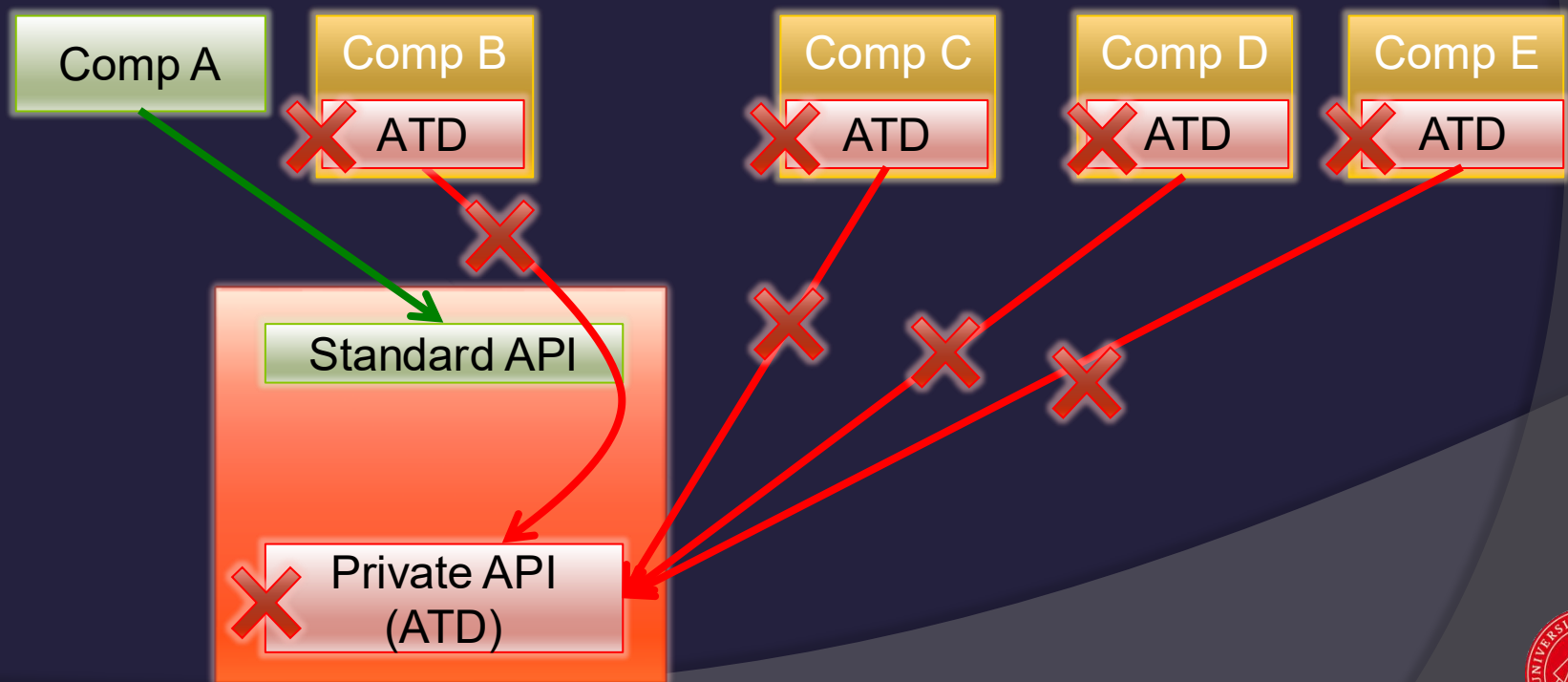


Growing cost of refactoring

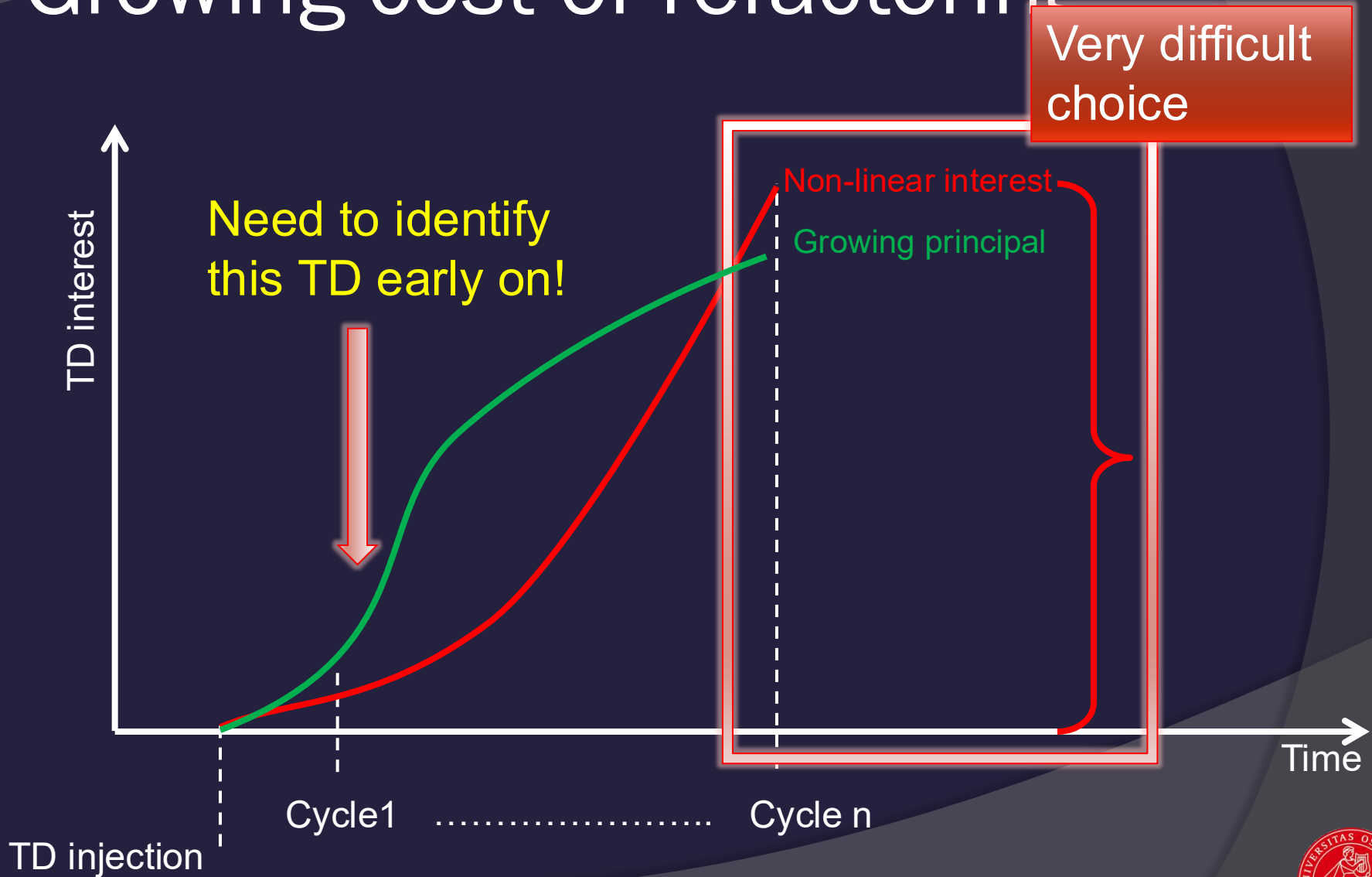


Growing refactoring costs

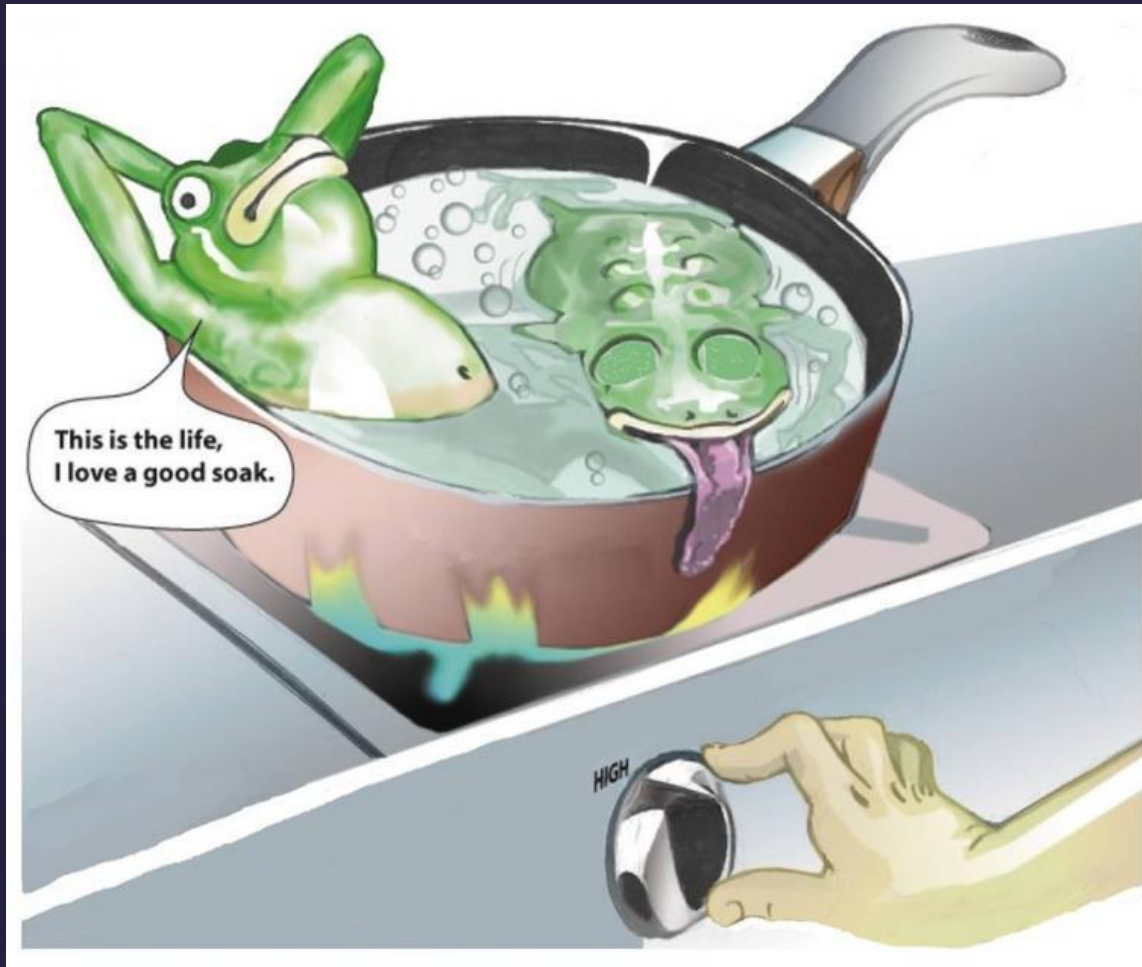
- *Example of TD and refactoring cost growing as the code grows*



Growing cost of refactoring

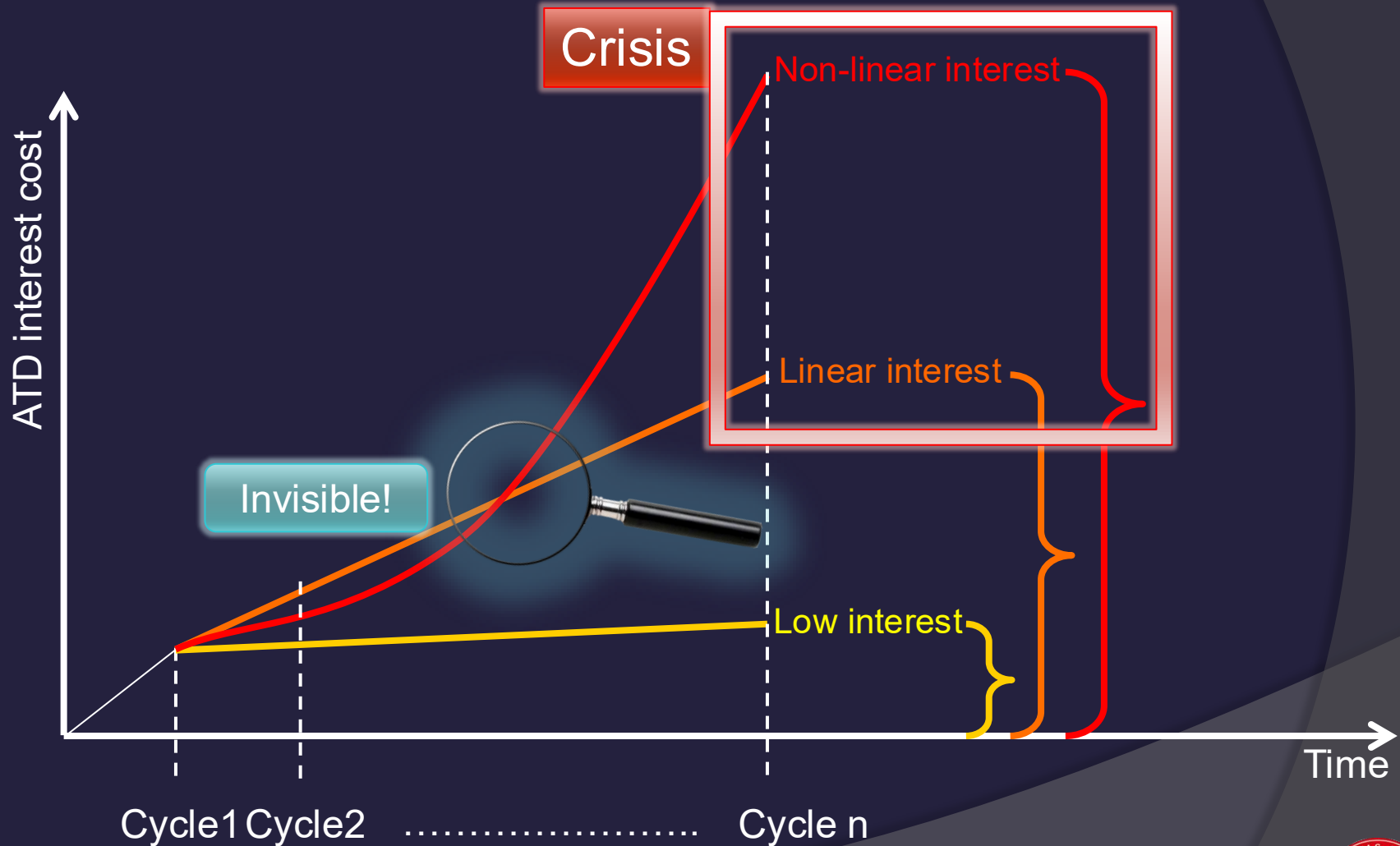


The boiling frog syndrome*



*Disclaimer: the effect doesn't seem to be true, but it's a good metaphor ☺

Growing interest is invisible!



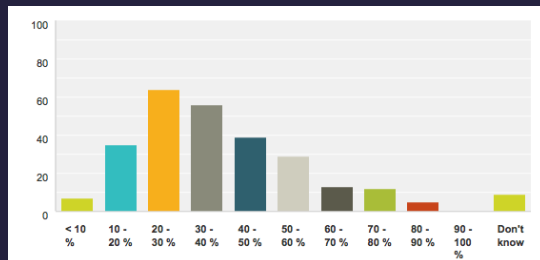
Martini, Bosch: "The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles,"
in *accepted for publication at WICSA 2015*, Montreal, Canada.

Antonio Martini - PhD in Software Engineering



Paying the bill: cost of TD

- 269 (+) respondents in 15 (+) organizations
- 43 developers reported over 3 months
 - Estimated **Interest (extra-costs)** of TD

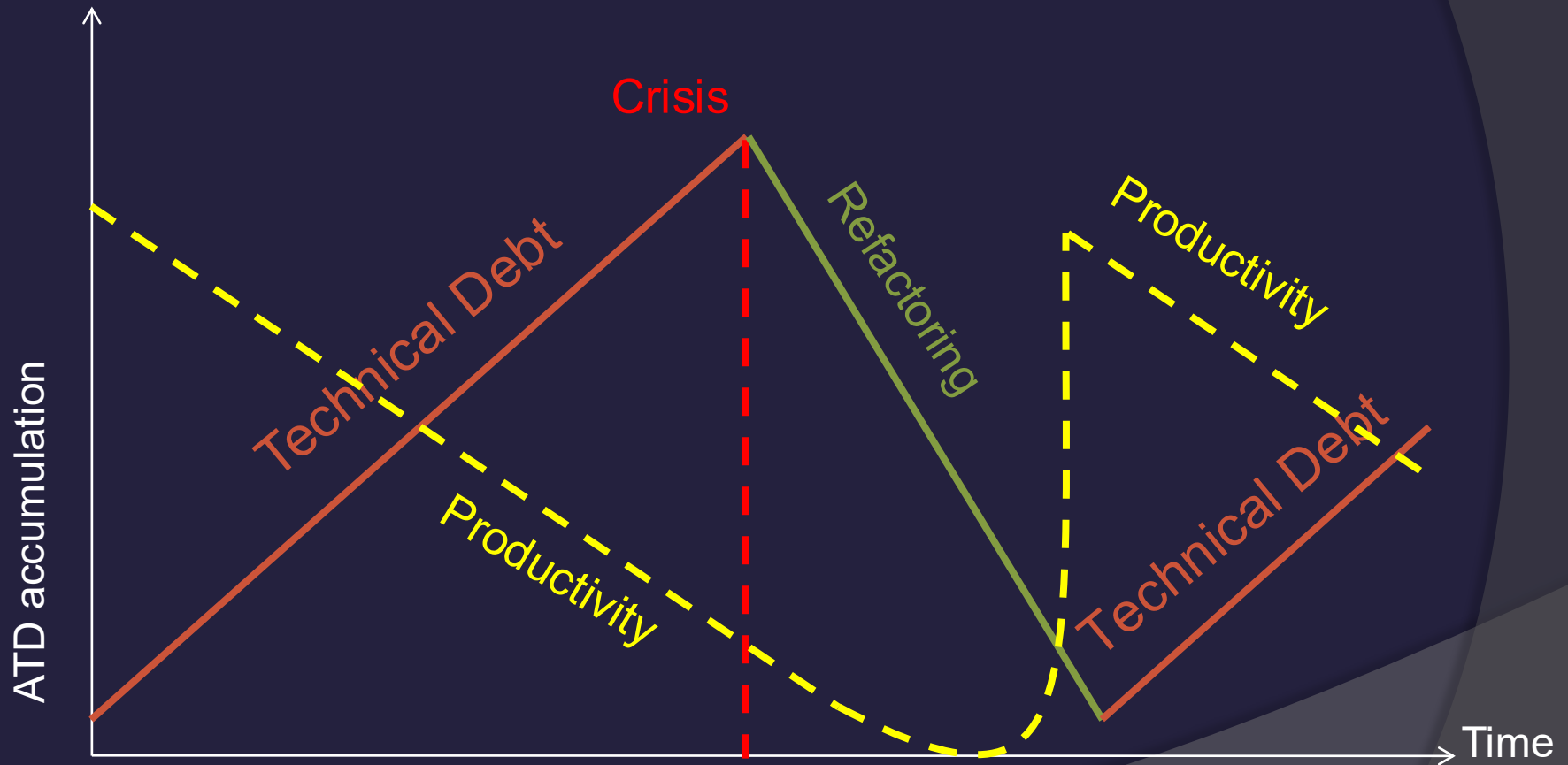


~ 30% waste

- Martini, Besker, Bosch: “*Introducing Technical Debt Tracking in Large Software Companies*”, APSEC 2016
- T. Besker, A. Martini, and J. Bosch, “The Pricey Bill of Technical Debt - When and by whom will it be paid?,” in Proceedings of ICSME 2017, Shanghai, China.

Why do we get Technical Debt?

Accumulation pattern for TD



* Martini, A., Bosch, J., Chaudron, M., 2015. "Investigating Architectural Technical Debt Accumulation and Refactoring over Time: Multiple-Case Study," *Information and Software Technology*.

Antonio Martini - PhD student in Software Engineering



Causes of accumulation?*

- ⦿ Business factors:
 - **Uncertainty** of use cases in the beginning
 - **Business evolution**
 - Time pressure: deadlines with penalties (**urgency**)
 - **Priority** of features over product architecture
 - **Split of budget** in Project and Maintenance
- ⦿ Lack of specification/emphasis on critical architectural requirements
- ⦿ Reuse of Legacy / third party / open source
- ⦿ **Parallel** development
- ⦿ Effects Uncertainty (**unknown** effects)
- ⦿ Non-completed Refactoring
- ⦿ **External** factors
 - Ex.: Technology evolution
- ⦿ Human factor

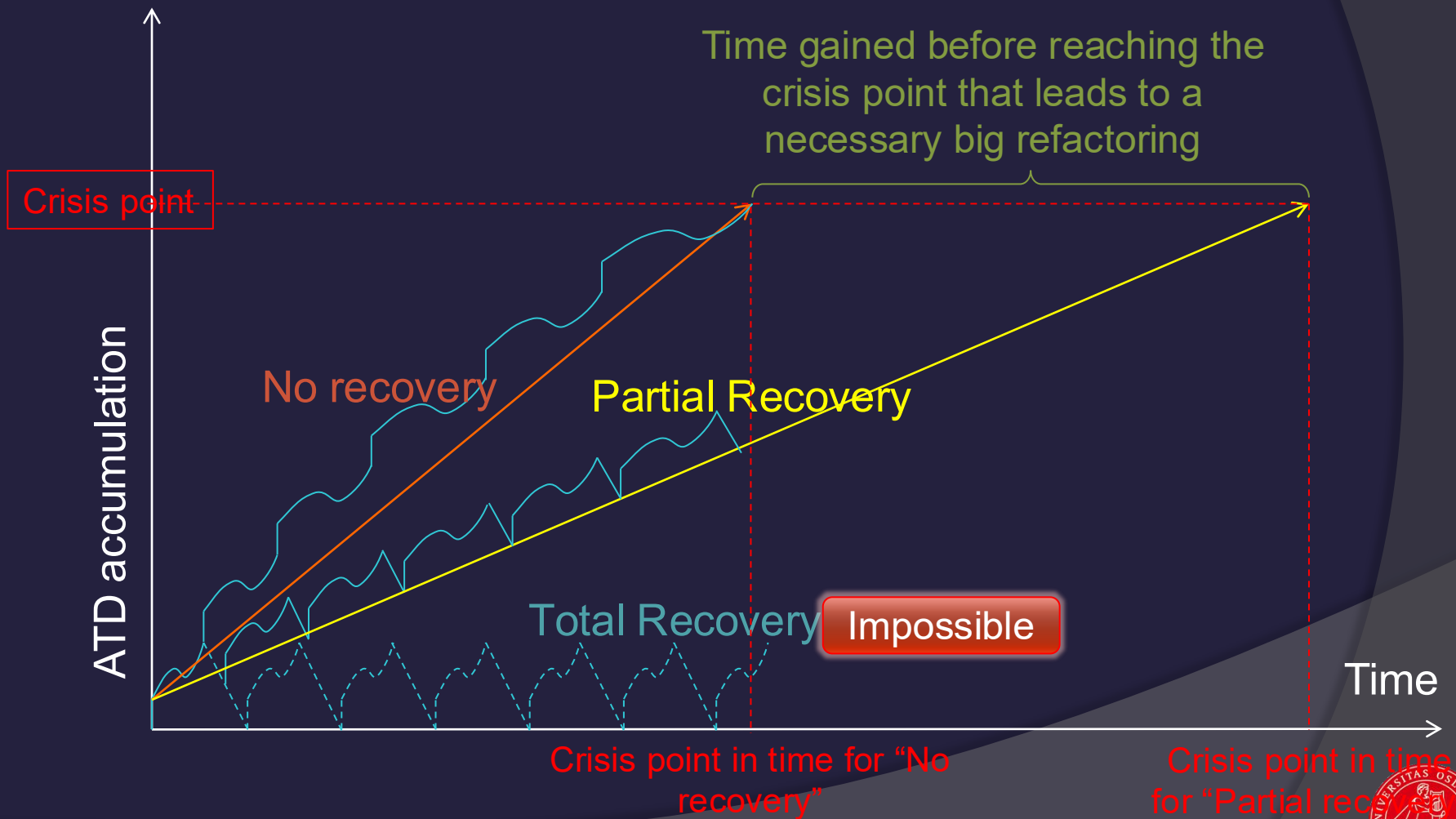
Impossible to
avoid TD

* Martini, A., Bosch, J., Chaudron, M., 2015. "Investigating Architectural Technical Debt Accumulation and Refactoring over Time: Multiple-Case Study," *Information and Software Technology*.



How can we reduce the number of crises?

Making the time-to-crisis longer?*



* Martini, A., Bosch, J., Chaudron, M., 2015. "Investigating Architectural Technical Debt Accumulation and Refactoring over Time: a Multiple-Case Study," *Information and Software Technology*.

Martini - PhD student in Software Engineering



Managing Technical Debt



Suggesting refactoring

- ⦿ During projects, we need to **refactor**
 - E.g. Removing technical debt
- ⦿ Refactoring means that we **change our system without changing its behavior.**
 - For example we make a method less complex but when we run the tests, the output doesn't change



«Smells»

Smelly but good



Smelly and not good



Code smells

Definition*:

A surface indication that usually corresponds to a deeper problem in the system

<https://martinfowler.com/bliki/CodeSmell.html>

Code smells

● Example

- Long parameter list

- Problems:

- It's not obvious what each int means at the call site.
- If you need to add more pricing-related parameters, the list gets worse.

```
// Code smell: too many related parameters
public int calculatePrice(int basePrice, int taxPercent, int discountPercent) {
    int priceWithTax = basePrice + (basePrice * taxPercent / 100);
    int finalPrice = priceWithTax - (priceWithTax * discountPercent / 100);
    return finalPrice;
}

// Usage
int price = calculatePrice(100, 20, 10);
```



Refactored code smell

● Refactoring:

● Parameter object

- The method has one clear parameter
- Related data «travels» together
- If you add another pricing-related field (e.g., shippingCost), you just add it to PriceParams without changing every call site's argument order.

```
// Group related parameters into a single class
class PriceParams {
    int basePrice;
    int taxPercent;
    int discountPercent;

    PriceParams(int basePrice, int taxPercent, int discountPercent) {
        this.basePrice = basePrice;
        this.taxPercent = taxPercent;
        this.discountPercent = discountPercent;
    }
}
```

```
public int calculatePrice(PriceParams params) {
    int priceWithTax =
        params.basePrice + (params.basePrice * params.taxPercent / 100);
    int finalPrice =
        priceWithTax - (priceWithTax * params.discountPercent / 100);
    return finalPrice;
}

// Usage
PriceParams params = new PriceParams(100, 20, 10);
int price = calculatePrice(params);
```



Should we always refactor code smells?

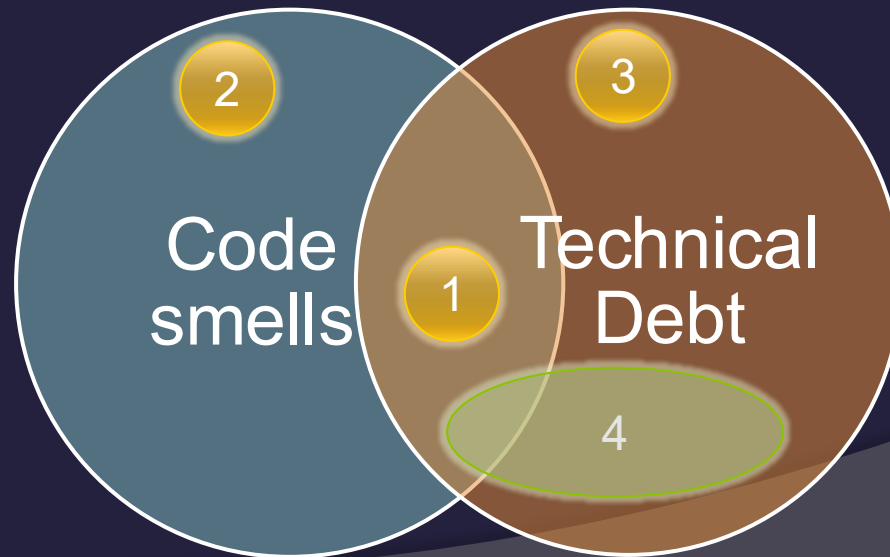
- No. Sometimes we might have code smells that are actually useful or not harmful

```
// Four parameters, but all are simple, obvious, and independent.  
void drawRectangle(int x, int y, int width, int height) {  
    // draw a rectangle at (x, y) with given width and height  
}
```

- Will a rectangle change?
- Is it a complex concept?
- Will we need to change this method?
- In most cases, the answer is no.
 - It doesn't make sense to refactor this code smell

Code smells and technical debt

1. Code smells can be an indication of technical debt
2. Some code smells are harmless
3. Some technical debt is not shown by code smells
4. Focus on the code-smells/technical debt that seem the most harmful



Example of tools

- SonarQube
- CodeScene



How do large companies manage TD?

- Companies manage TD with different **levels of maturity***

* Martini, Besker, Bosch: “*Introducing Technical Debt Tracking in Large Software Companies*”
accepted at APSEC 2016 and in press



Level 0 - Unaware



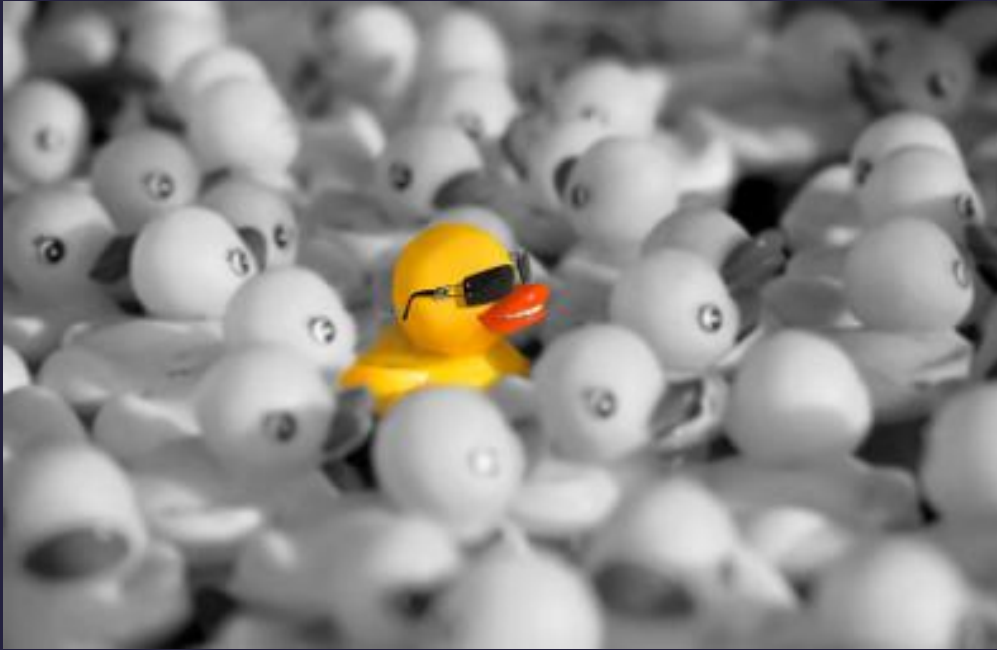
- There is no awareness of TD in the organization

Level 1 - No Tracking



- We know what TD is...
- ...but we don't do much in our organization...

Level 2 – Ad-hoc tracking



- No budget allocated
- Driven by individual initiative
- Use of “improper” tools



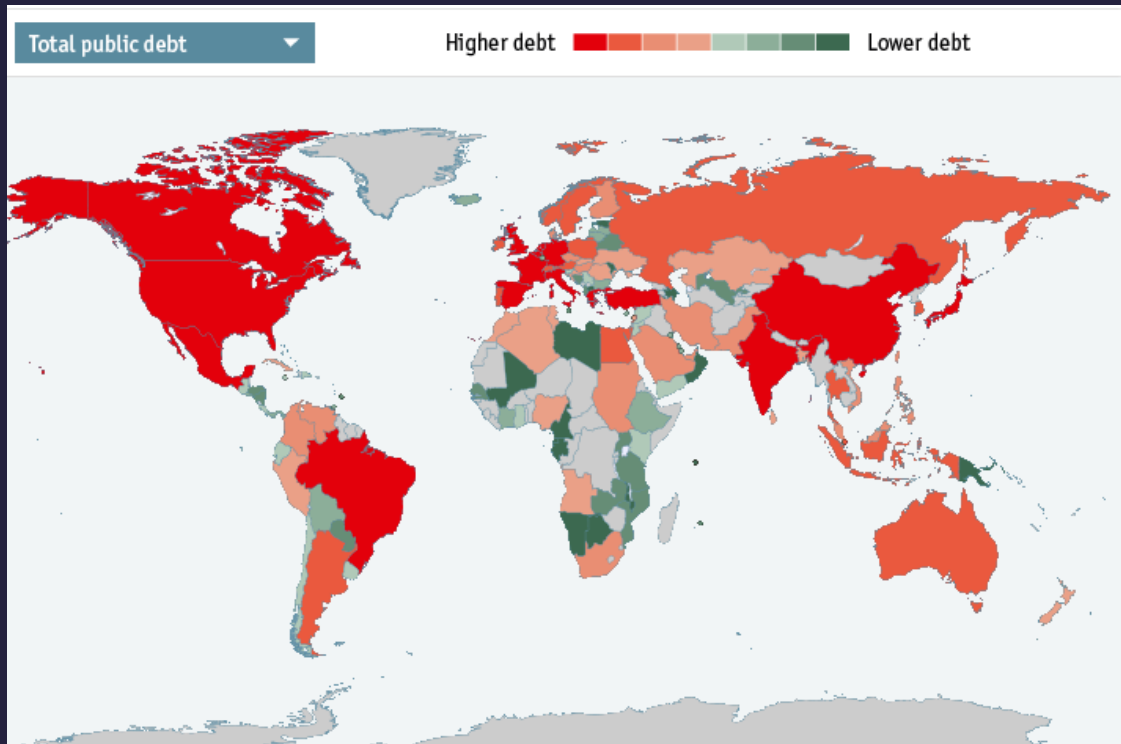
Level 3 – Systematic tracking



- ◉ Budget allocated to TD management
- ◉ TD-specific documentation
- ◉ Iterative process to monitor TD issues
- ◉ Continuous TD process improvement



Level 4 – Measured



- Use of automated TD identification tools
- Use of automated indicators to measure the interest of TD
- Integration of several measurements

Level 5 – Institutionalized



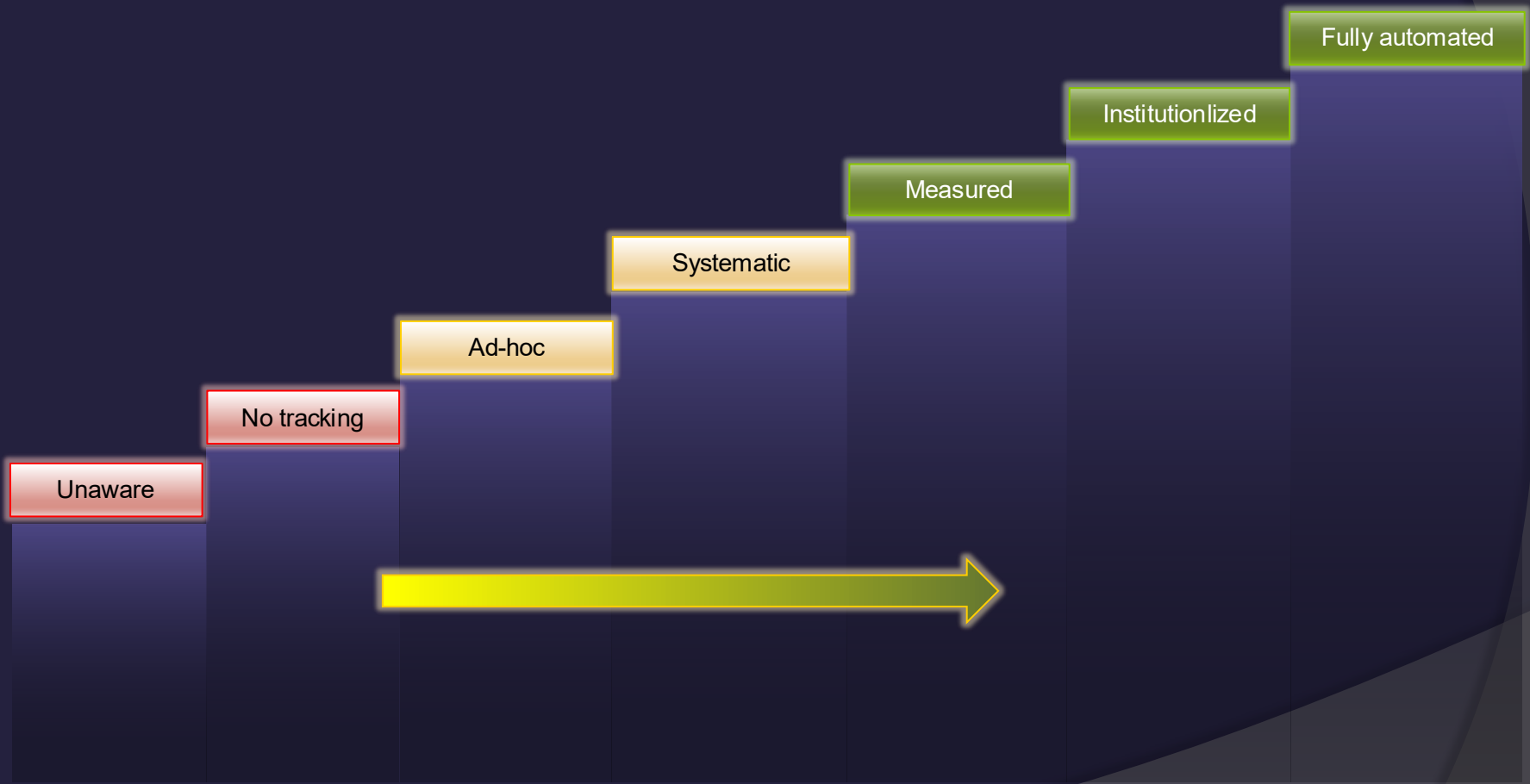
- Process standardized across the organization
- Spread to the whole organization
- Used for lifecycle planning
- Prioritization done among different kinds of TD

Level 6 – Fully automated



- ⦿ Automated data-driven decisions
- ⦿ statistical data from the history of the system
- ⦿ assessment with existing benchmarks
- ⦿ (Still a research work in progress)

Goal about Technical Debt Management*

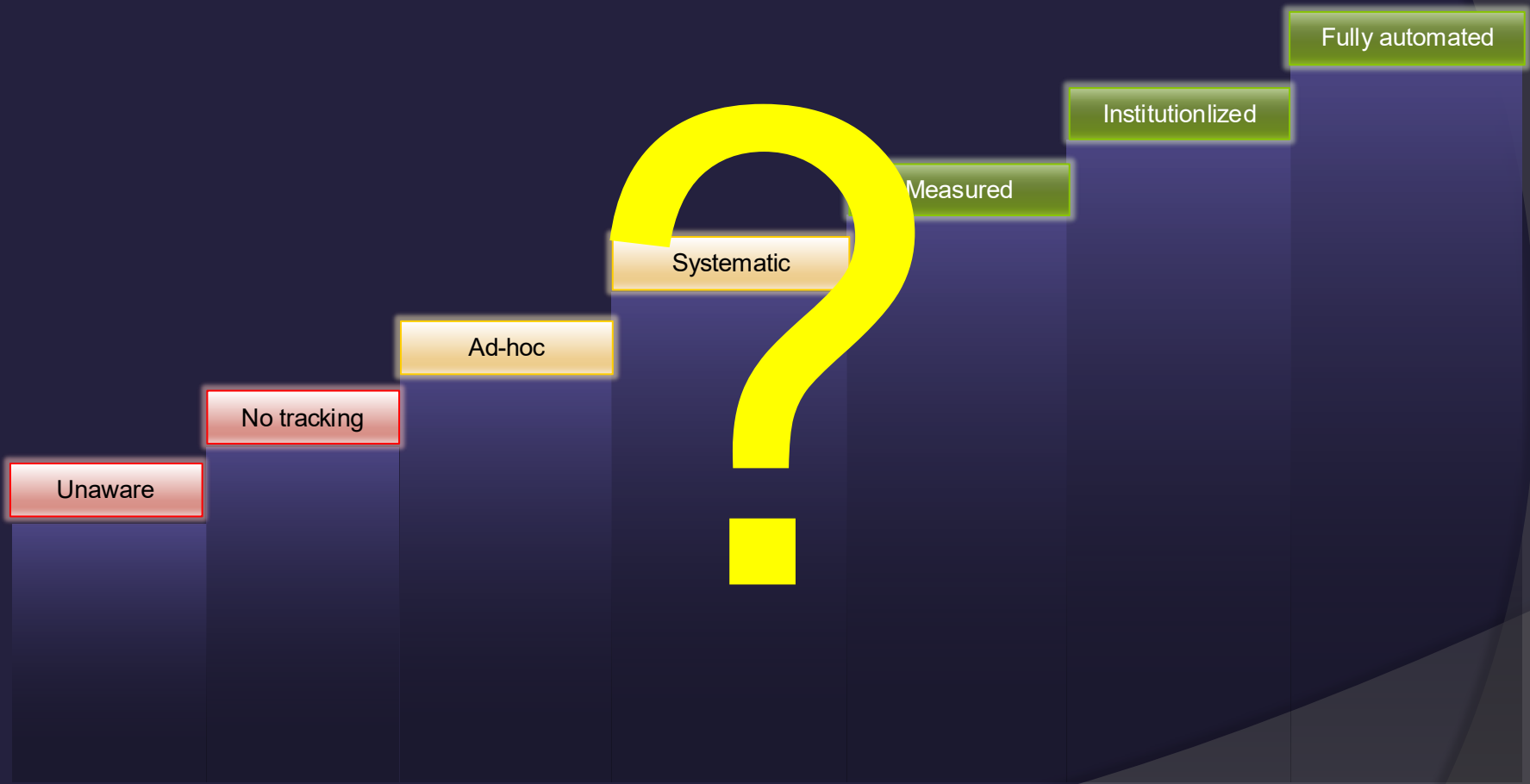


* Martini, Antonio; Besker, Terese & Bosch, Jan (2018). Technical Debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations. *Science of Computer Programming*.

Antonio Martini - PhD in Software Engineering



Where are you now?

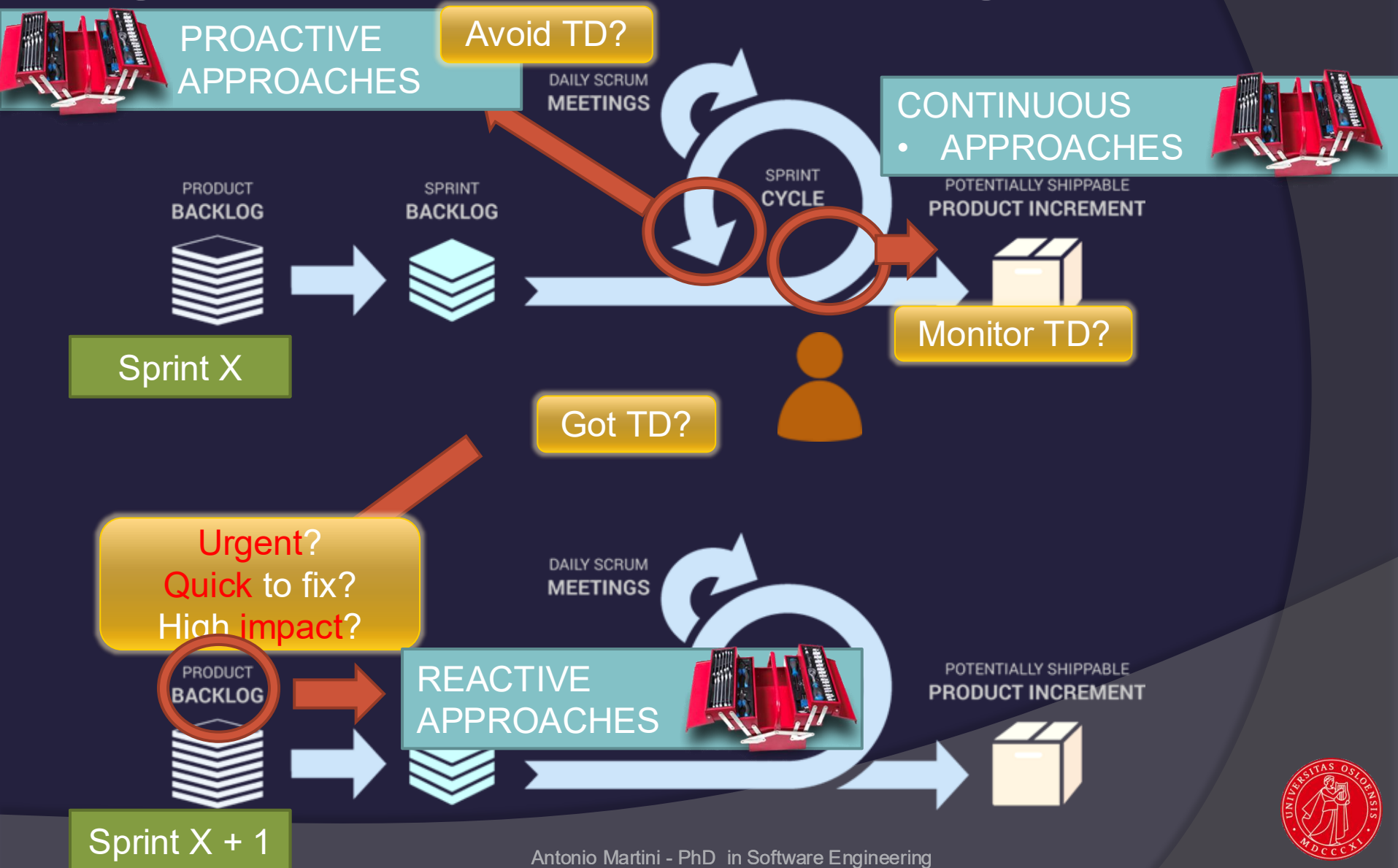


* Martini, Antonio; Besker, Terese & Bosch, Jan (2018). Technical Debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations. *Science of Computer Programming*.

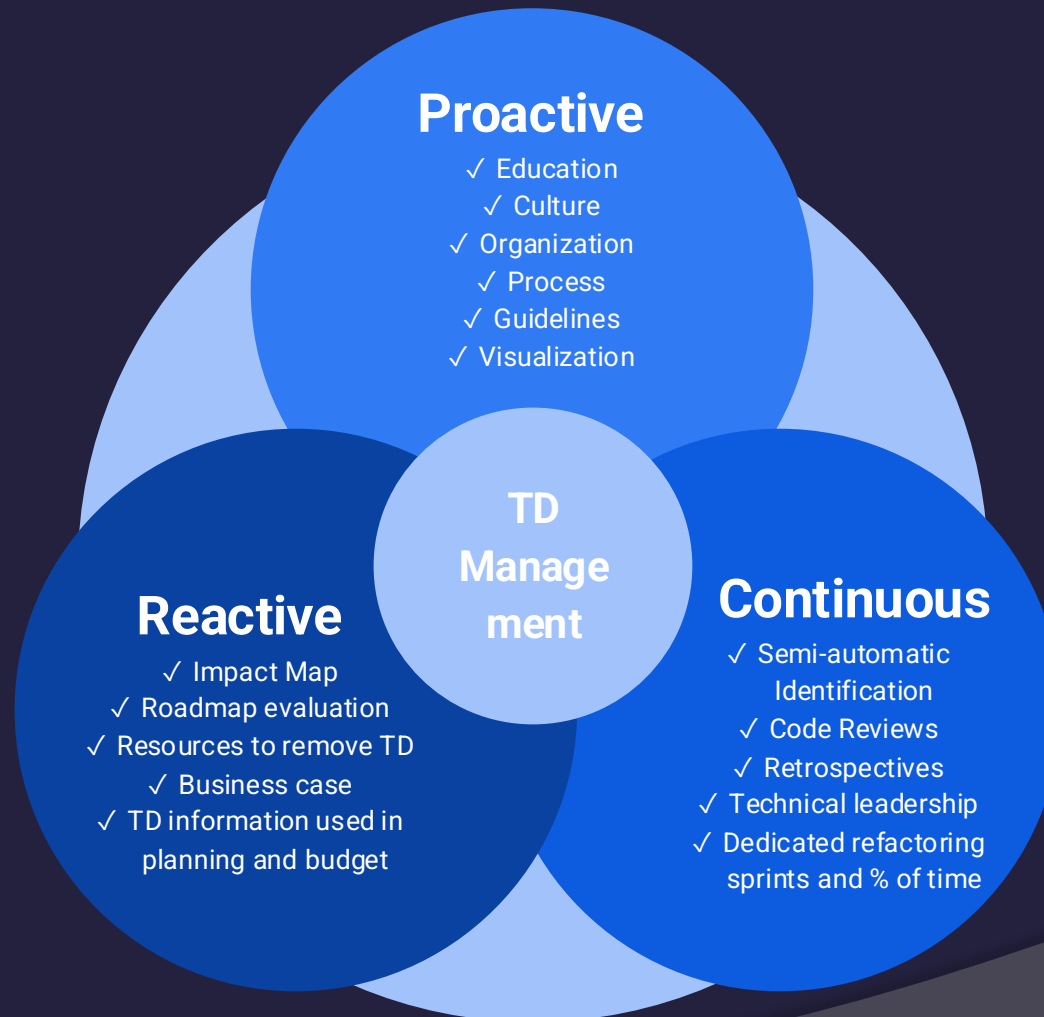
Antonio Martini - PhD in Software Engineering



Agile Technical Debt management



Technical Debt Management covers several aspects



Proactive approaches

- Education
- Culture
- Organization
- Process
- Guidelines
- **Visualization**

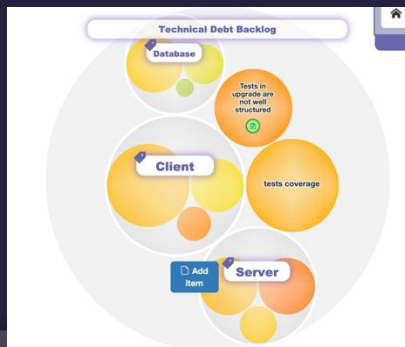
Seminars or workshops about **how to** manage TD?

Does everyone **encourage** avoiding and removing TD?

Who is **responsible** for TD?

Is there an **iterative process** to assess TD?

Do we **know what** and **how** to assess and fix?



Education

- ◎ Raise **awareness** of Technical Debt for everyone in a company
 - Facilitate
 - **Seminars**
 - **Working sessions**
 - To:
 - Understand the concepts
 - Work with existing issues
- ◎ We have experience that **education works!**
 - 67 respondents in a large company:
 - The seminar was



Useful

4.4

Culture and mindset

⦿ Encouragement *:

- Make the developers feel more **in control**
- **reduces** the negative impact of technical debt on developers' morale **



⦿ **Managers** and **peers** are key for encouragement:

- to consider TD important
- without encouragement, nothing works



* Terese Besker, Antonio Martini, and Jan Bosch. 2020. Carrot and stick approaches when managing technical debt. In *Proceedings of the 3rd International Conference on Technical Debt (TechDebt '20)*. Association for Computing Machinery, New York, NY, USA, 21–30

** Terese Besker, Hadi Ghanbari, Antonio Martini, Jan Bosch, The influence of Technical Debt on software developer morale, *Journal of Systems and Software*, 2020

Continuous approaches

- Semi-automatic Identification
- Code Reviews
- Retrospectives
- Technical leadership
- Dedicated refactoring sprints and % of time

Do we have **static code analyzers**?



Do we have a **Champion** who takes **initiative**?



■ Development ■ Technical Debt



Refactoring time: example

- ◉ We plan 5 sprints:
 - Case 1:
 - we plan to develop 5 features
 - What about Technical Debt? if we have time, we fix it...
 - Results: we don't have time
 - Case 2:
 - We plan 4 features
 - We leave 20% time to fix Technical Debt
 - Should we dedicate 20% for each sprint for TD?
 - Should we have 1 full sprint dedicated to TD?



Which approaches do you use? 20% spread out or 1/5 dedicated sprint?



Refactoring time: example - 2

- It depends:

- A **larger** technical debt item requires a sprint
 - Sometimes, if it's really big (e.g. architecture) it might require an epic
- If we need to clean up **smaller** TD, 20% over time can be better

- Important: planning refactoring time helps

- Expectation management**
- Avoiding **unplanned** work and **stress**



Reactive approaches

- Impact Map
- Roadmap evaluation
- Resources to remove TD
- Business case
- TD information used in planning and budget

What and **who** is TD going to **impact**?



Do we have **enough** and the **right** people to handle TD?



We should **expect** TD



Assessing Technical Debt Management

Technical Debt cannot be fully avoided, it can be useful but needs to be controlled

Don't wait until it's
too late



Focus on removing
the debt with the
most interest

TD management is a learning process

Practices



TD pratice suites

TDMA - Visma

Assessment of Capabilities



Assessment Tools

TD-Pulse Analysis

Results



KPIs

TD-Pulse Analysis

Assessing the **strategic management** of Technical Debt*

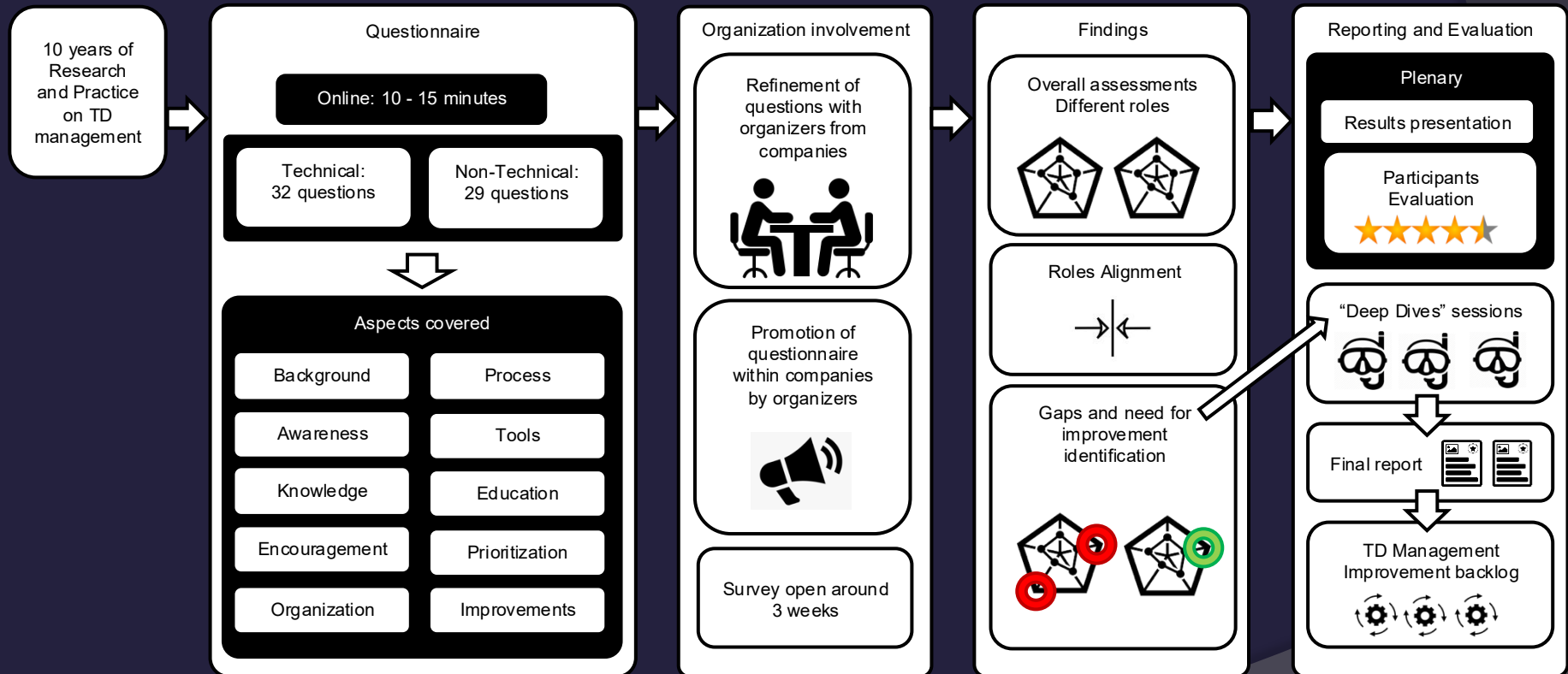
- ◎ **TD-Pulse*** method
- ◎ Used with:
 - 417 technical and non-technical roles
 - 7 large companies

<https://ieeexplore.ieee.org/document/9976055>

*

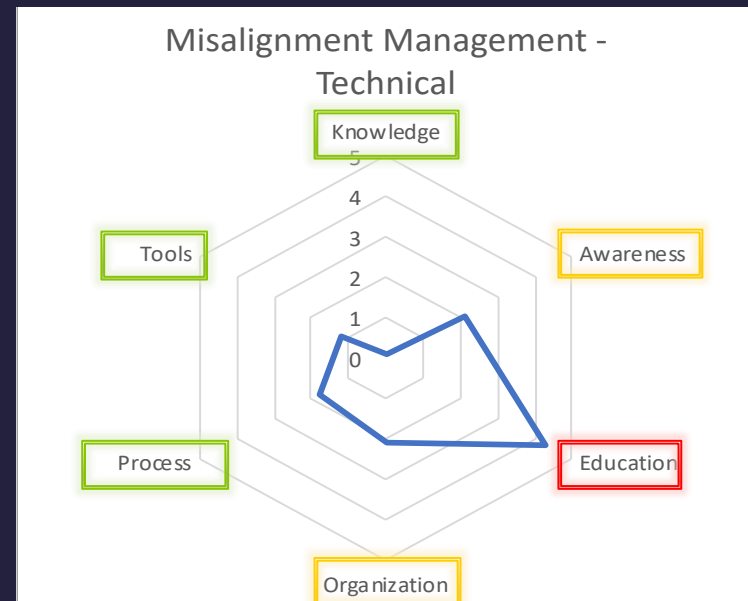
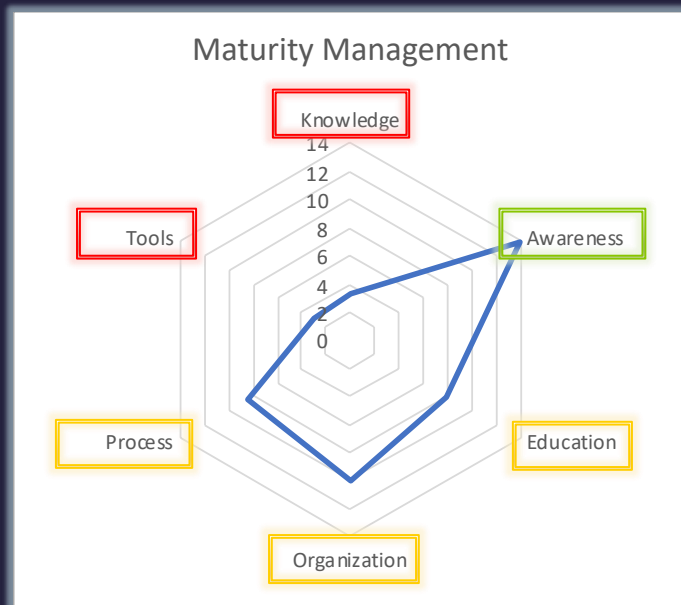


TD Pulse approach*



<https://ieeexplore.ieee.org/document/9976055>

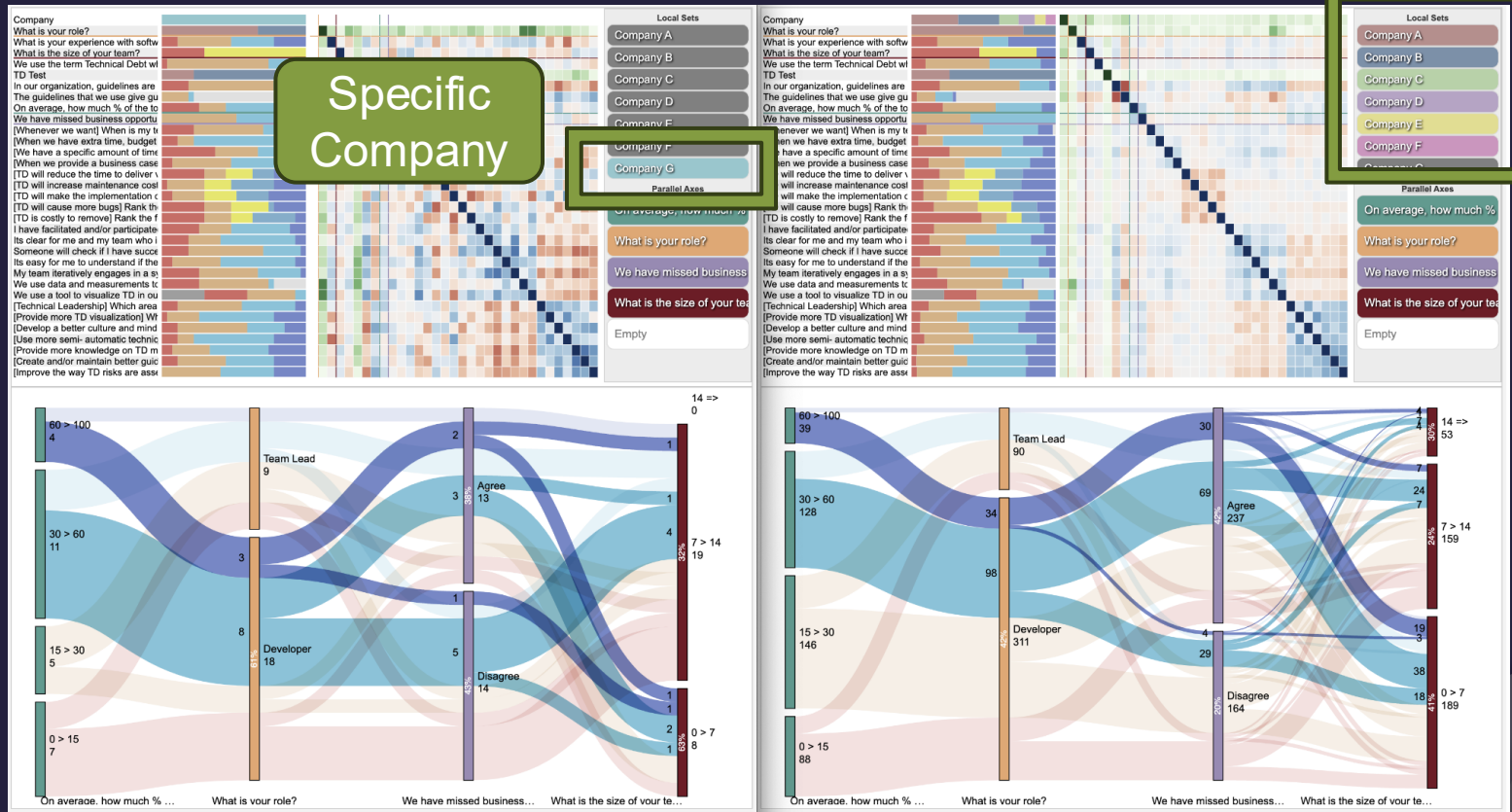
Example of high-level assessment*



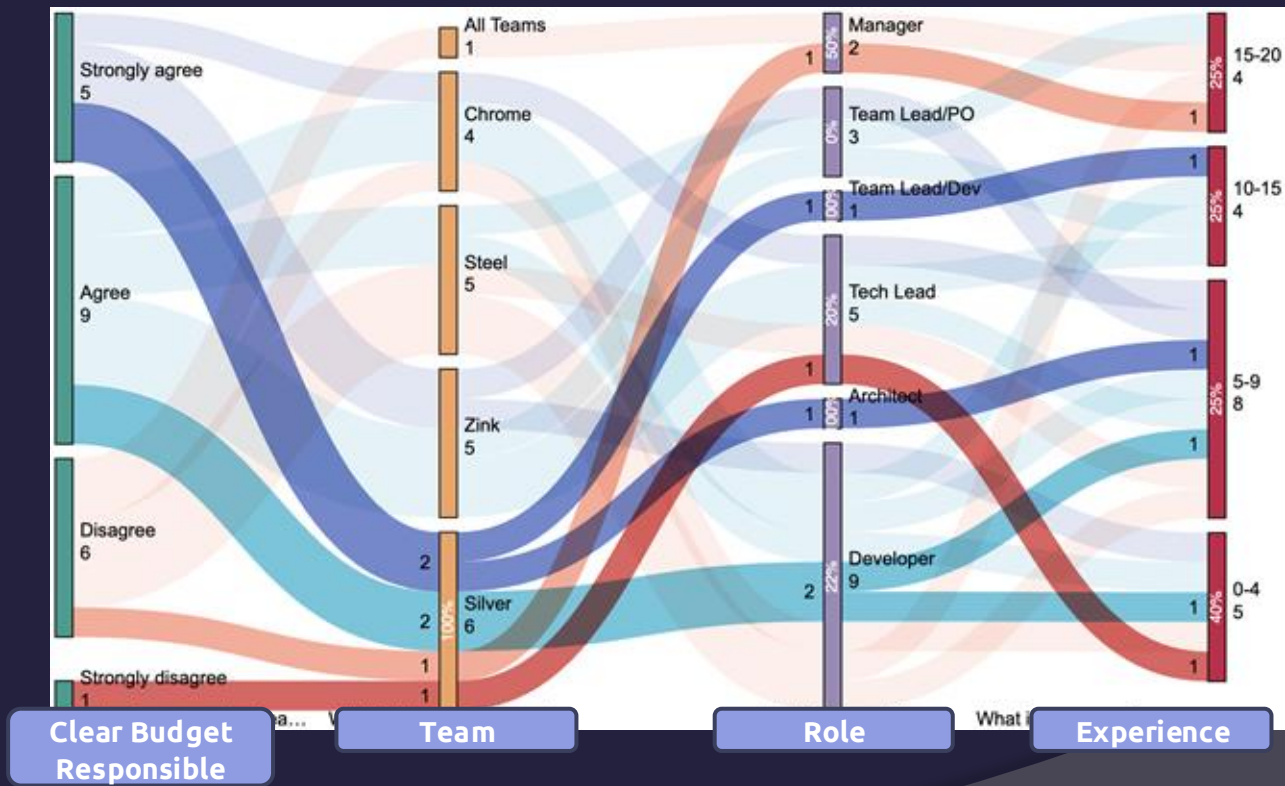
<https://ieeexplore.ieee.org/document/9976055>

Discover Misalignments Across Organizations (benchmarking)

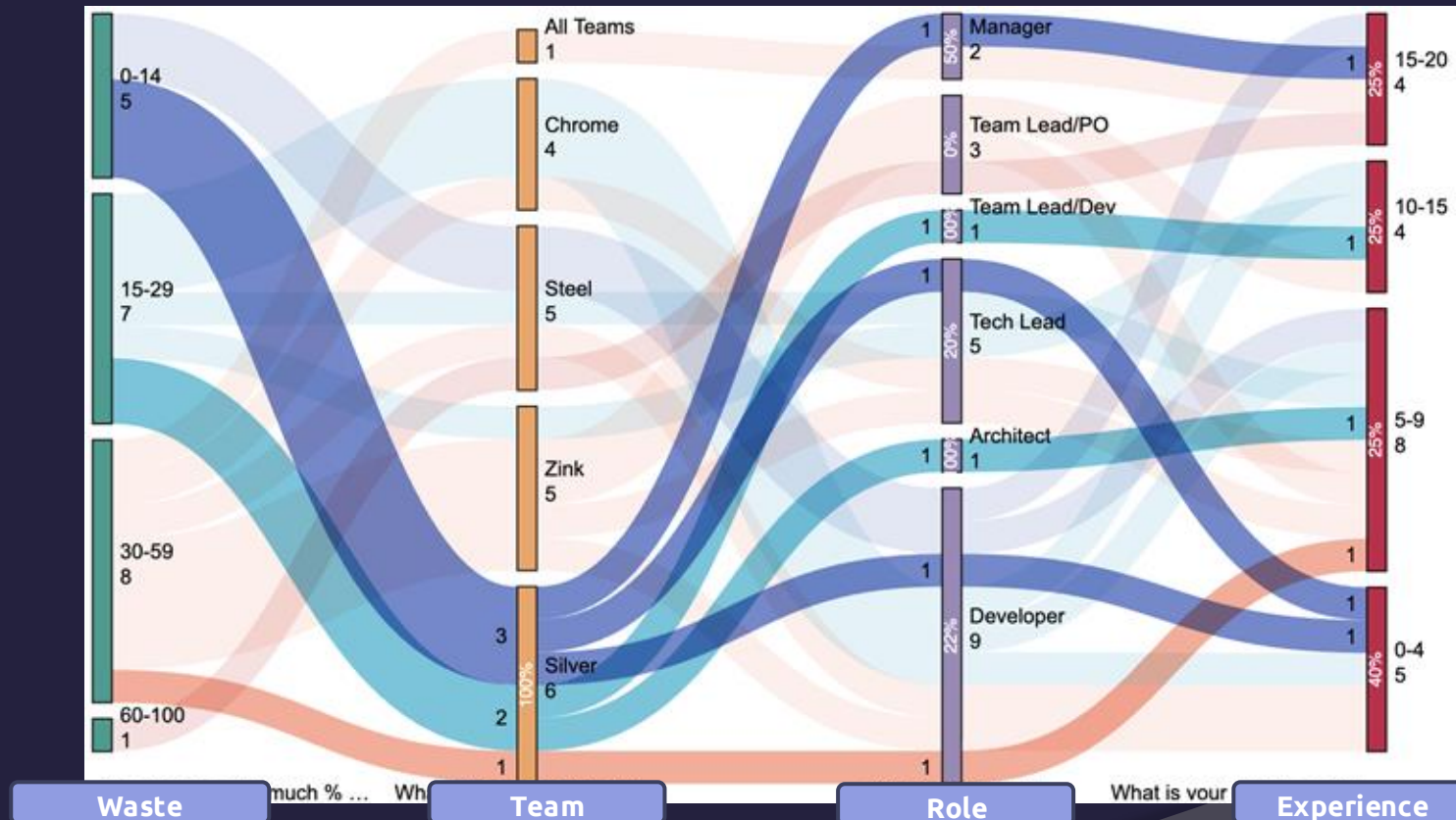
Other Companies



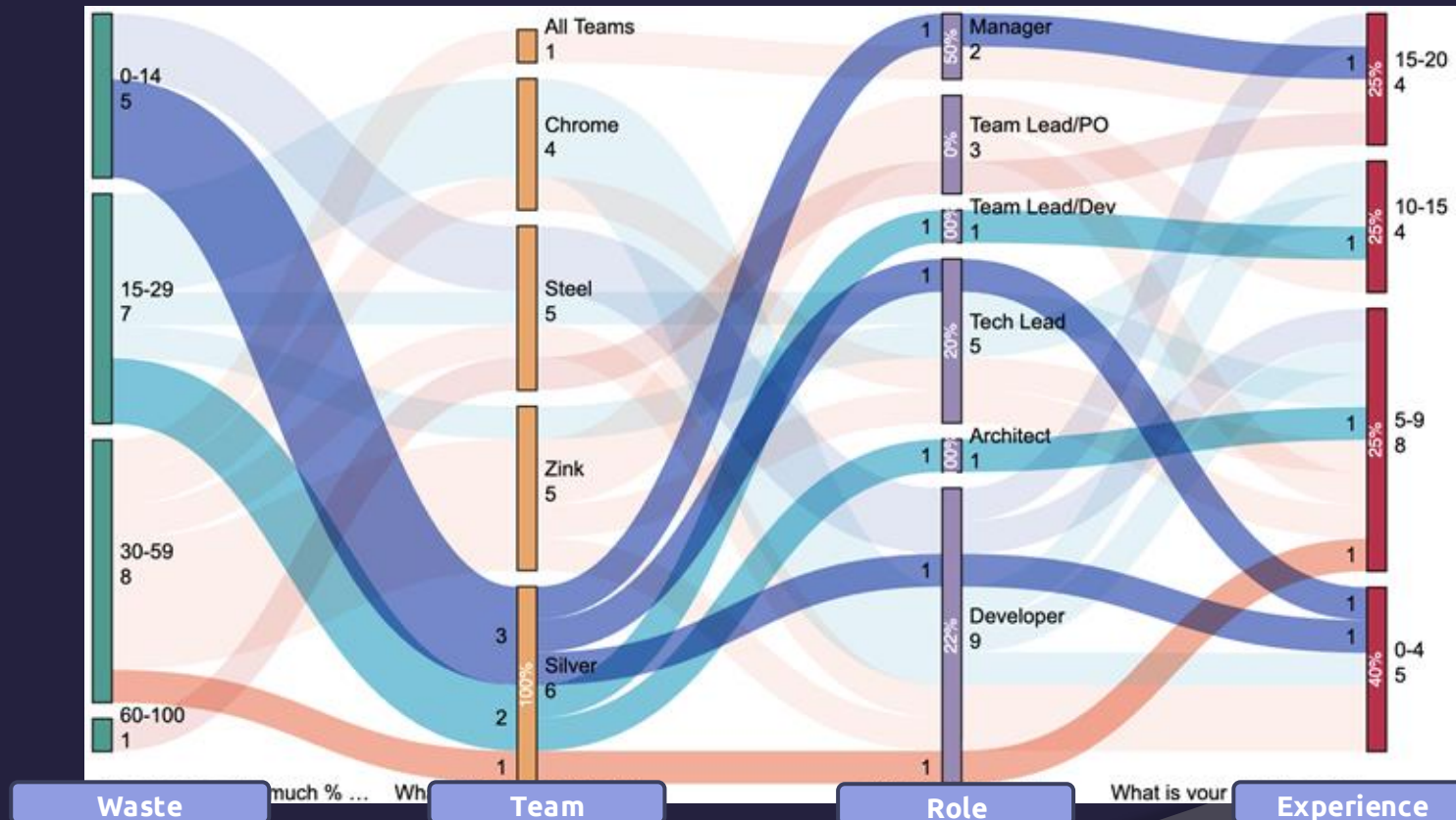
Pinpoint Misalignments Within Teams



Misalignment about the harmfulness of TD (KPI)



Misalignment about the harmfulness of TD (KPI)



TD management metrics vs TD code metrics

*«Would you have been able to find these insights **using traditional code-based** approaches?»*

Company A:

*«Very likely **no**, I would say, because this is more about **learning the perception of the organization**. So these are **very useful insights**. I view it as pretty **orthogonal** to code-based measurements»*



Antonio Martini



- **Professor of Software Engineering at UiO**
- Group Leader for the SE Group
- Technical Debt, AI, Agile, Architecture, AI
- Previously
 - Software Developer
 - PhD and Postdoc in Software Engineering (Chalmers)
 - Independent Consultant
 - Principal, Strategic Researcher at CA Technologies (now Broadcom)
 - Founder of 2 Startups
- <https://www.linkedin.com/in/antmartini>

Maren Maritsdatter Kruke



- **Security Business Analyst, Visma**
- Facilitating secure software development
- Working with ~200 companies within the Visma Group
- Research projects
- <https://www.linkedin.com/in/marenmaritsdatterkruke>

Mili Orucevic



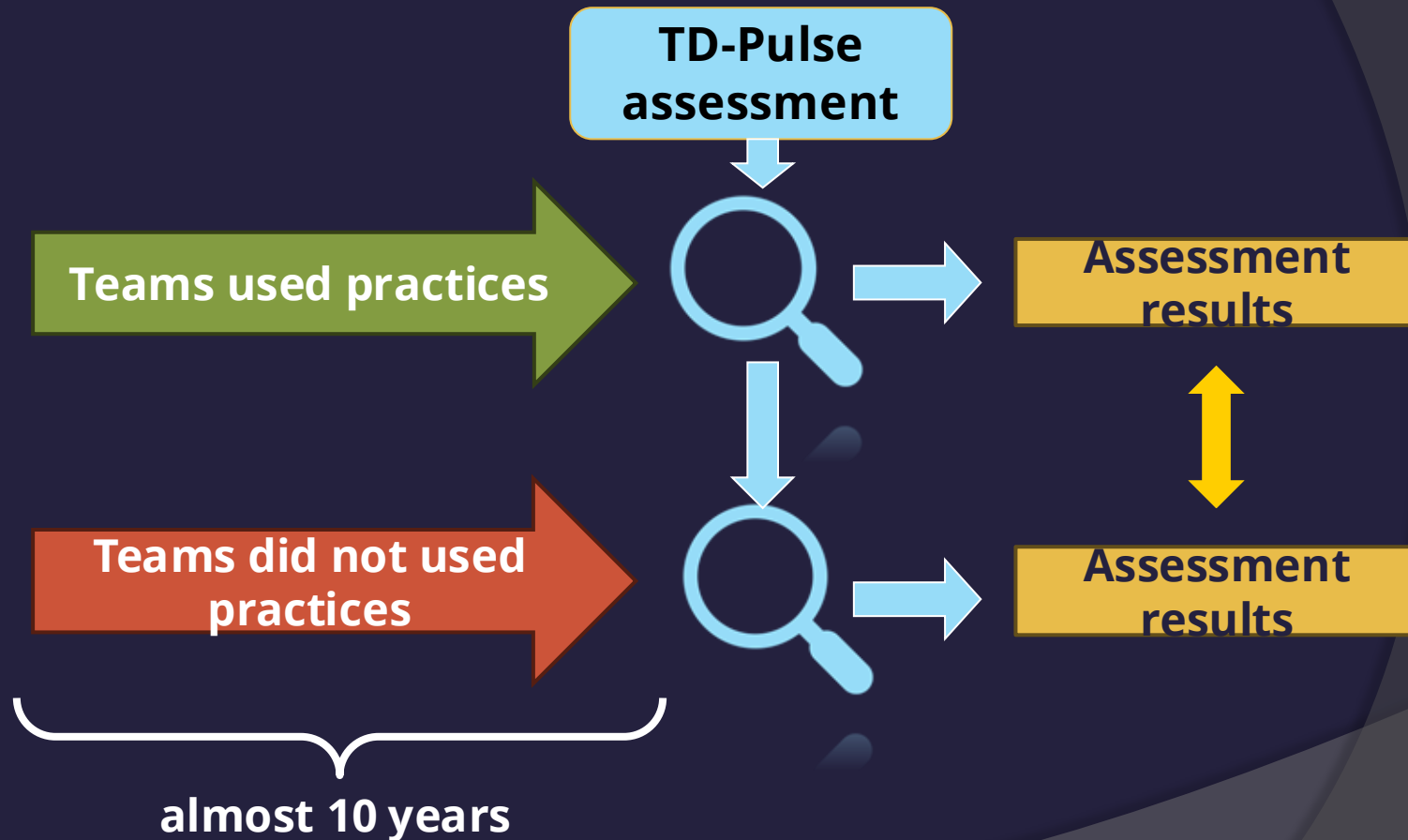
- **AI Tech Lead, Visma**
- AI-native Software Development
- Digital transformations to modern software engineering, cloud, and AI
- Working with ~200 companies within the Visma Group
- Research projects
- <https://www.linkedin.com/in/milio>



UNIVERSITETET
I OSLO



Technical Debt Management: Study at Visma (91 teams)



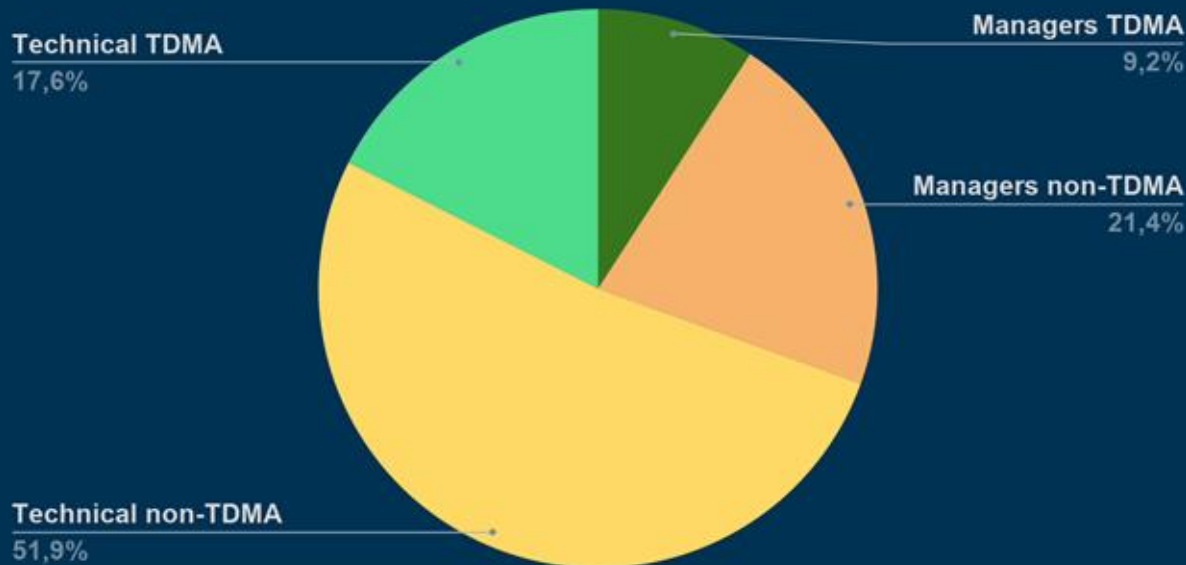
Participants on TD-Pulse at Visma

- Questionnaire: online Sep 11 – Oct 2, 2023
- 40 Management Roles
- 91 Technical Roles



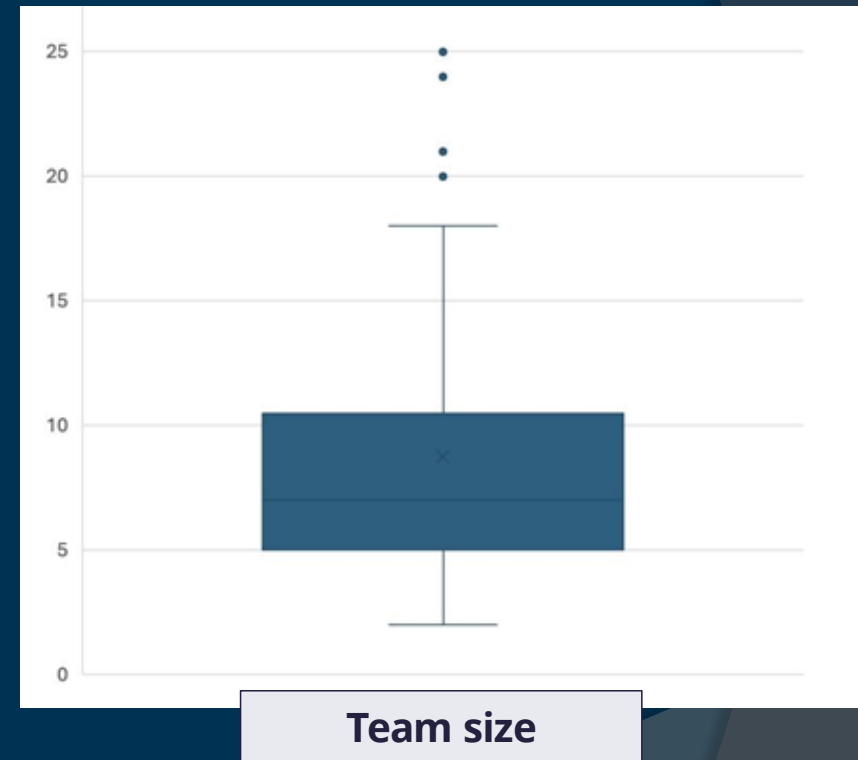
Technical Roles

Managers

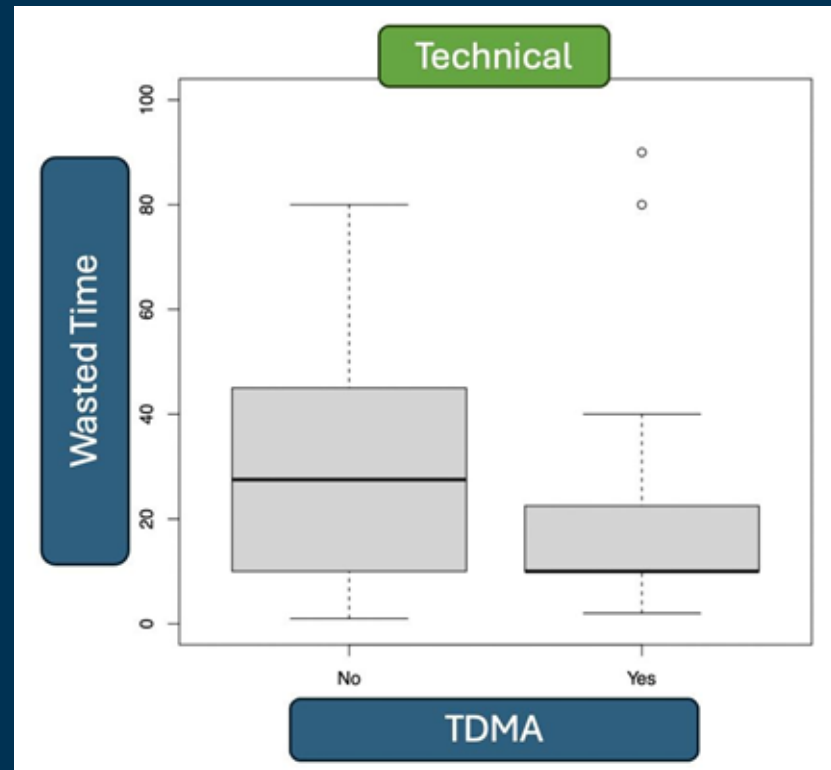
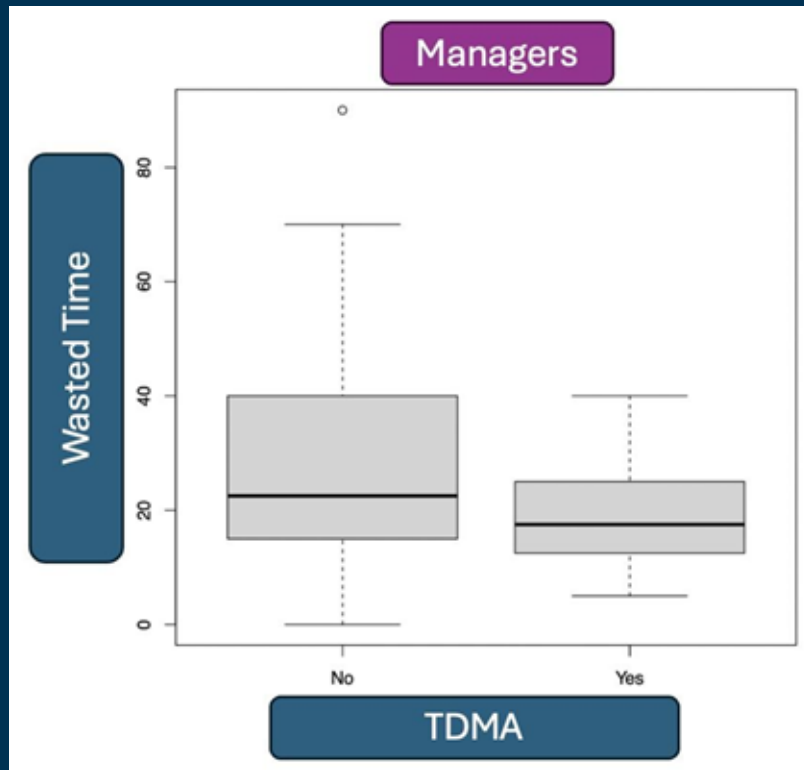


Category	Count
Managers	
Service/Product Owner	14
Team Manager	14
Other Management	12
Technical	
Developer	55
Architect	23
Other Technical Roles	13

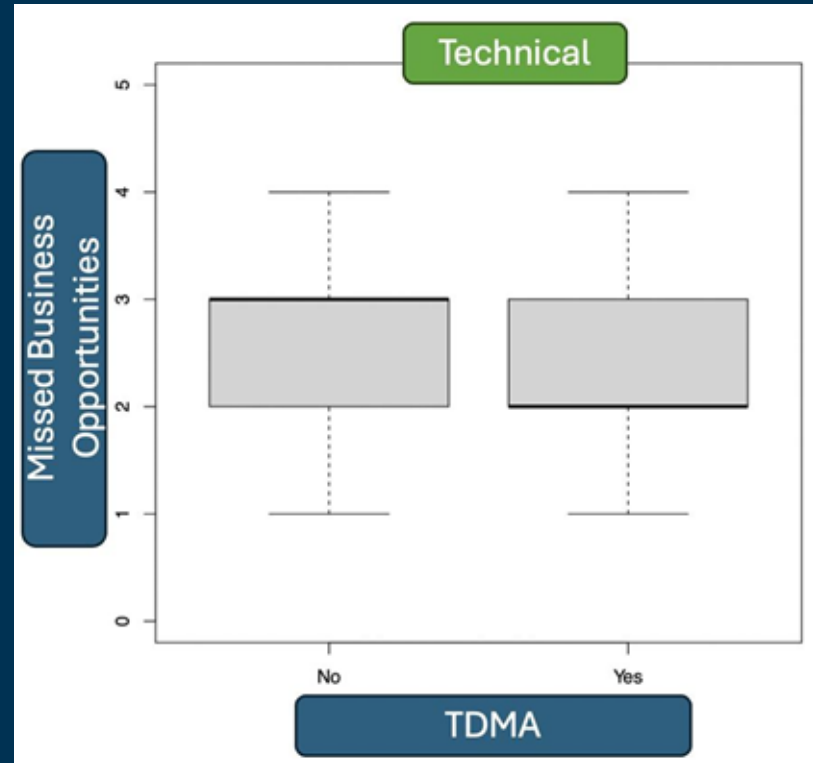
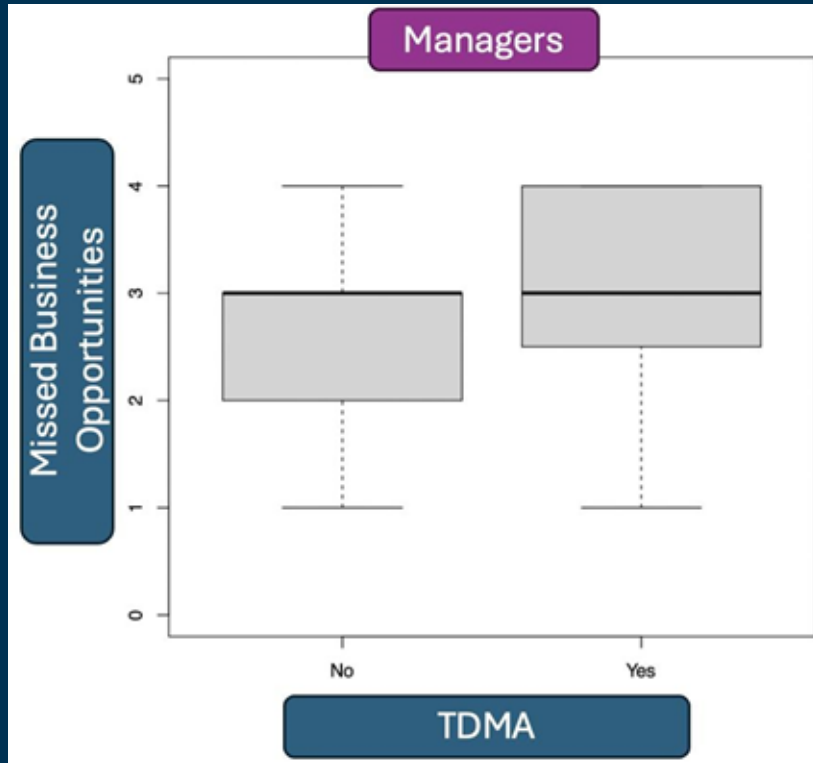
Demographics of participants



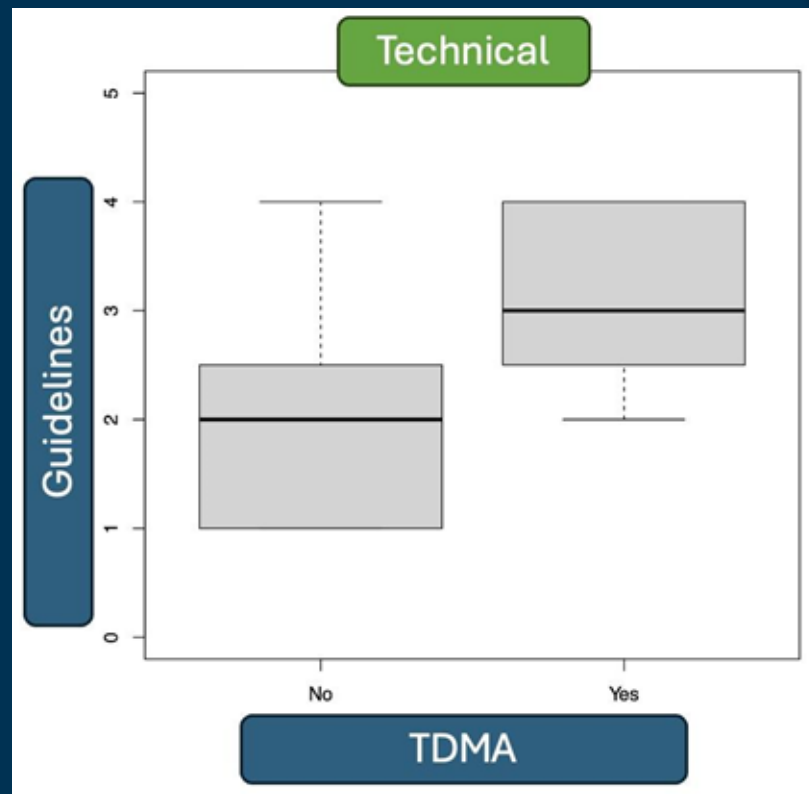
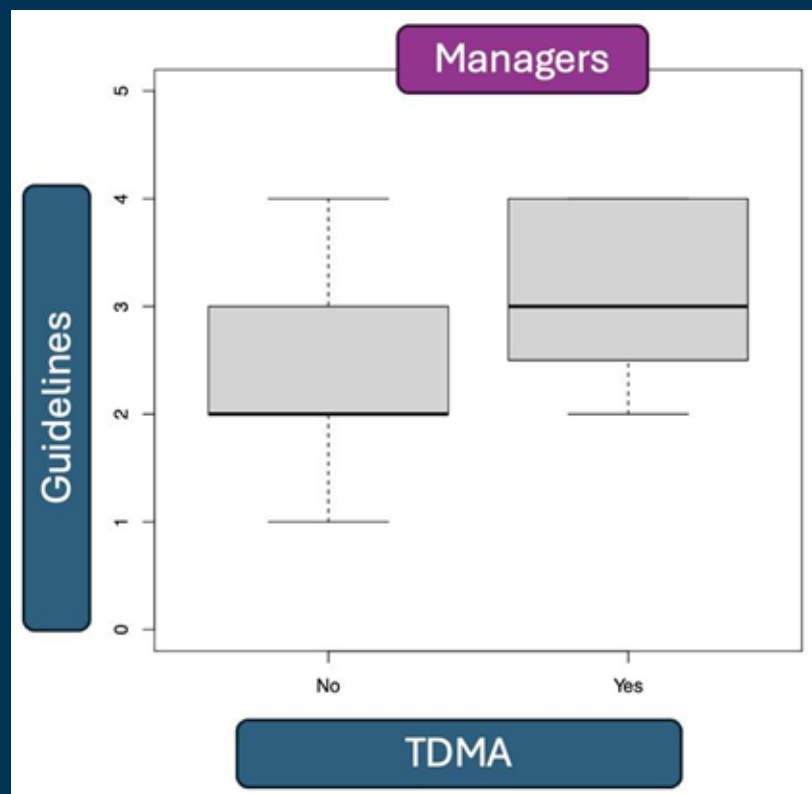
TDMA reduced the perception of wasted time



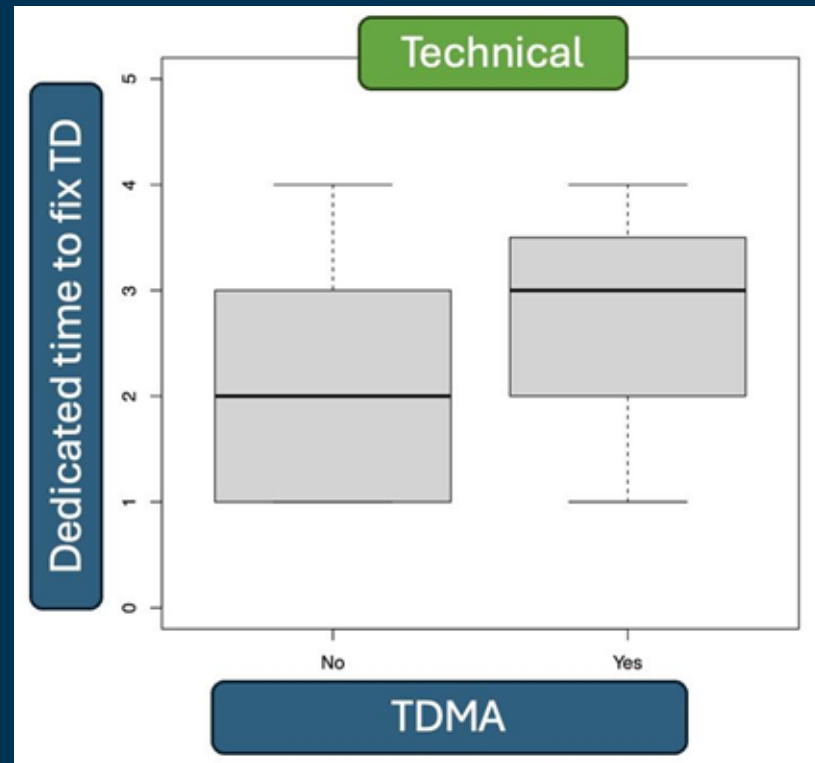
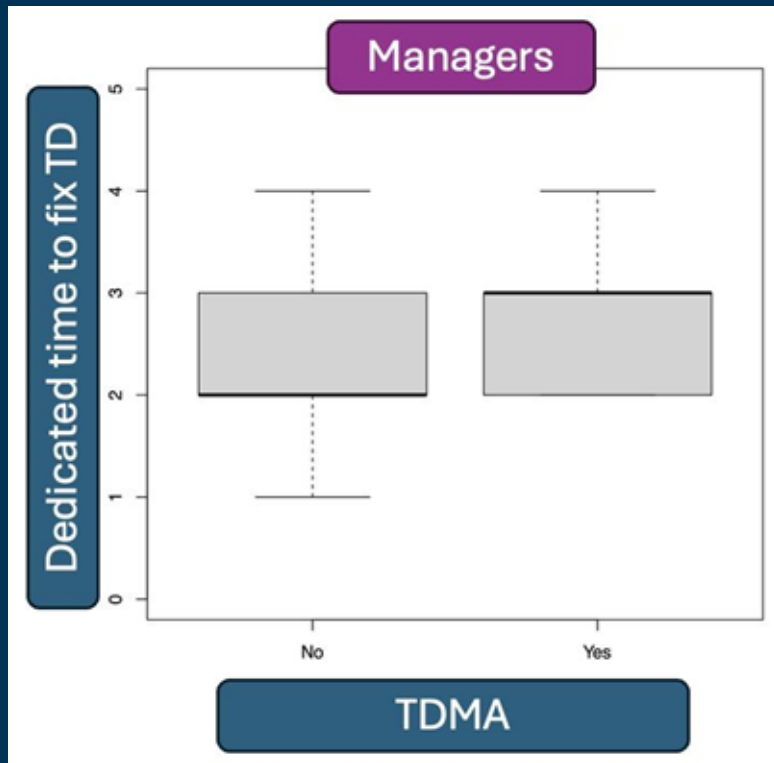
TDMA does not associate with less missed business opportunities



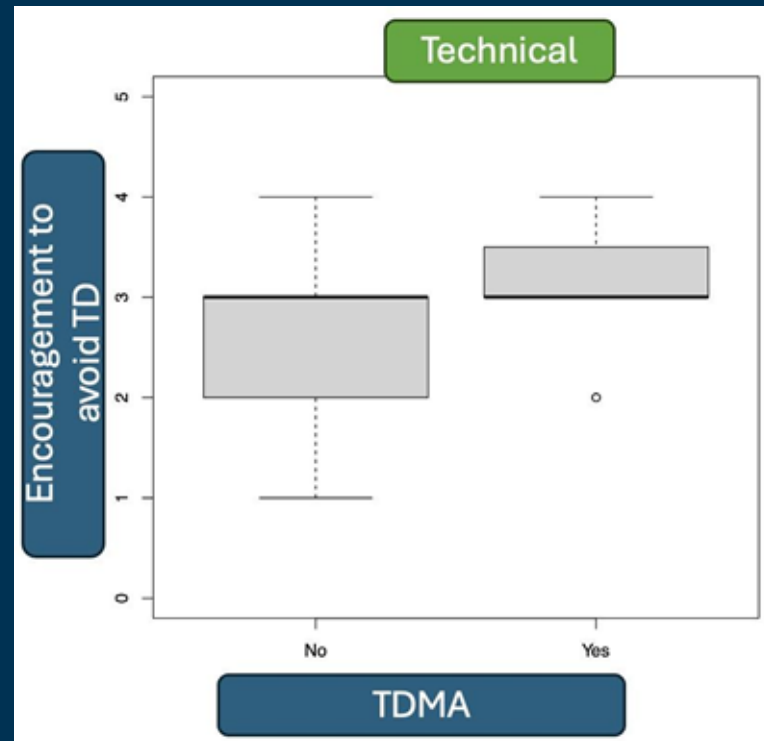
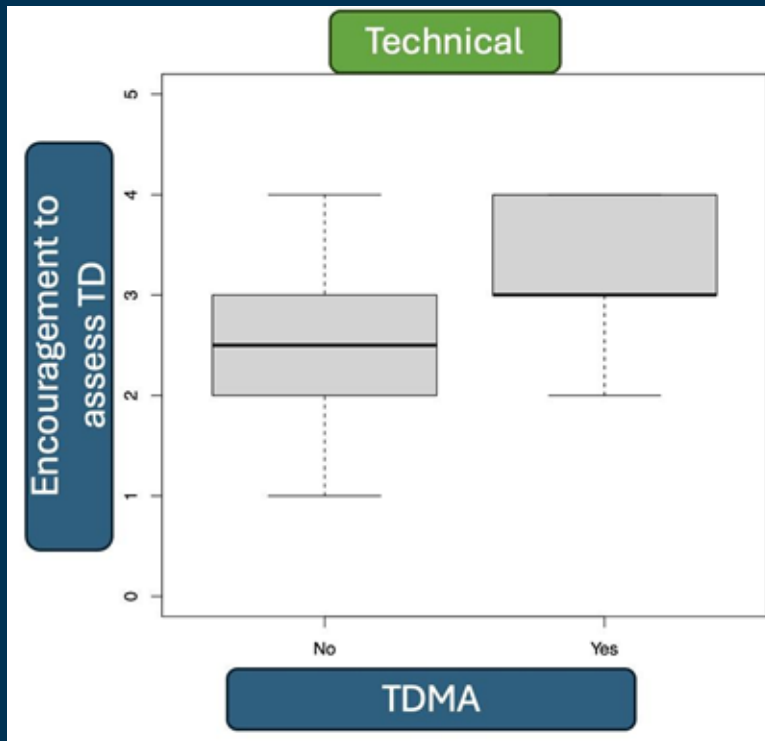
TDMA associates with better guidelines



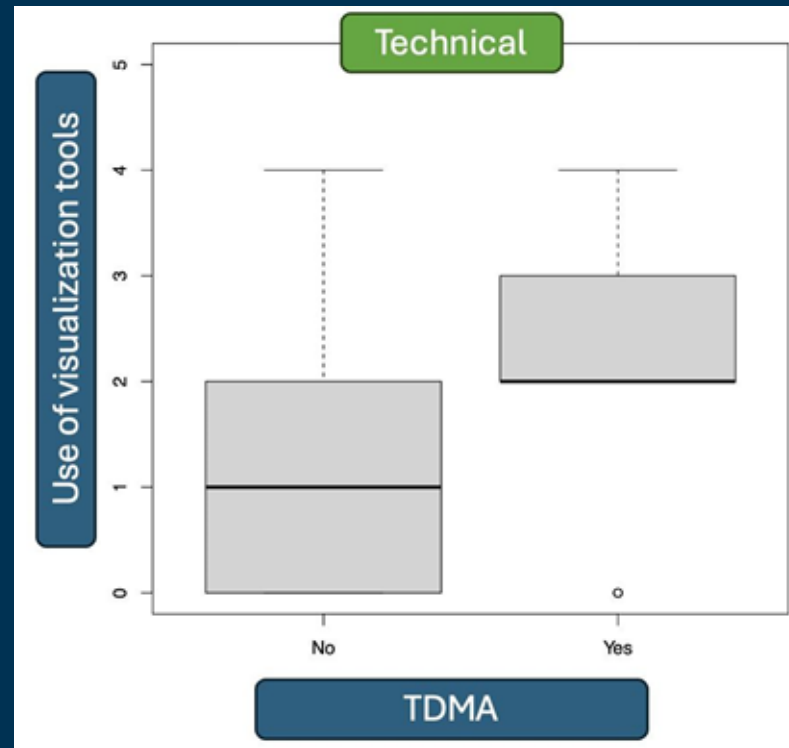
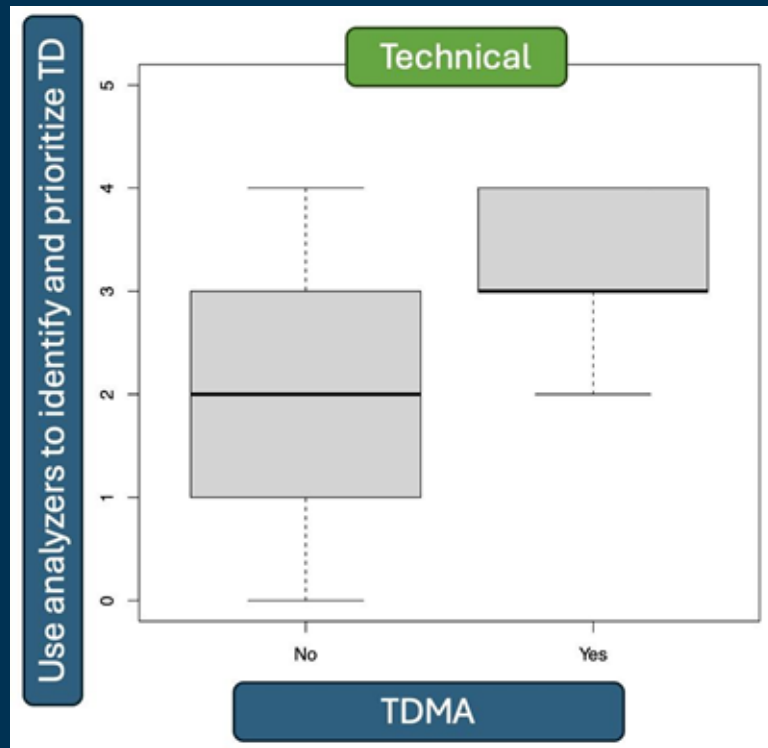
TDMA associates with more dedicated time to fix TD



TDMA associates with more technical roles encouraged



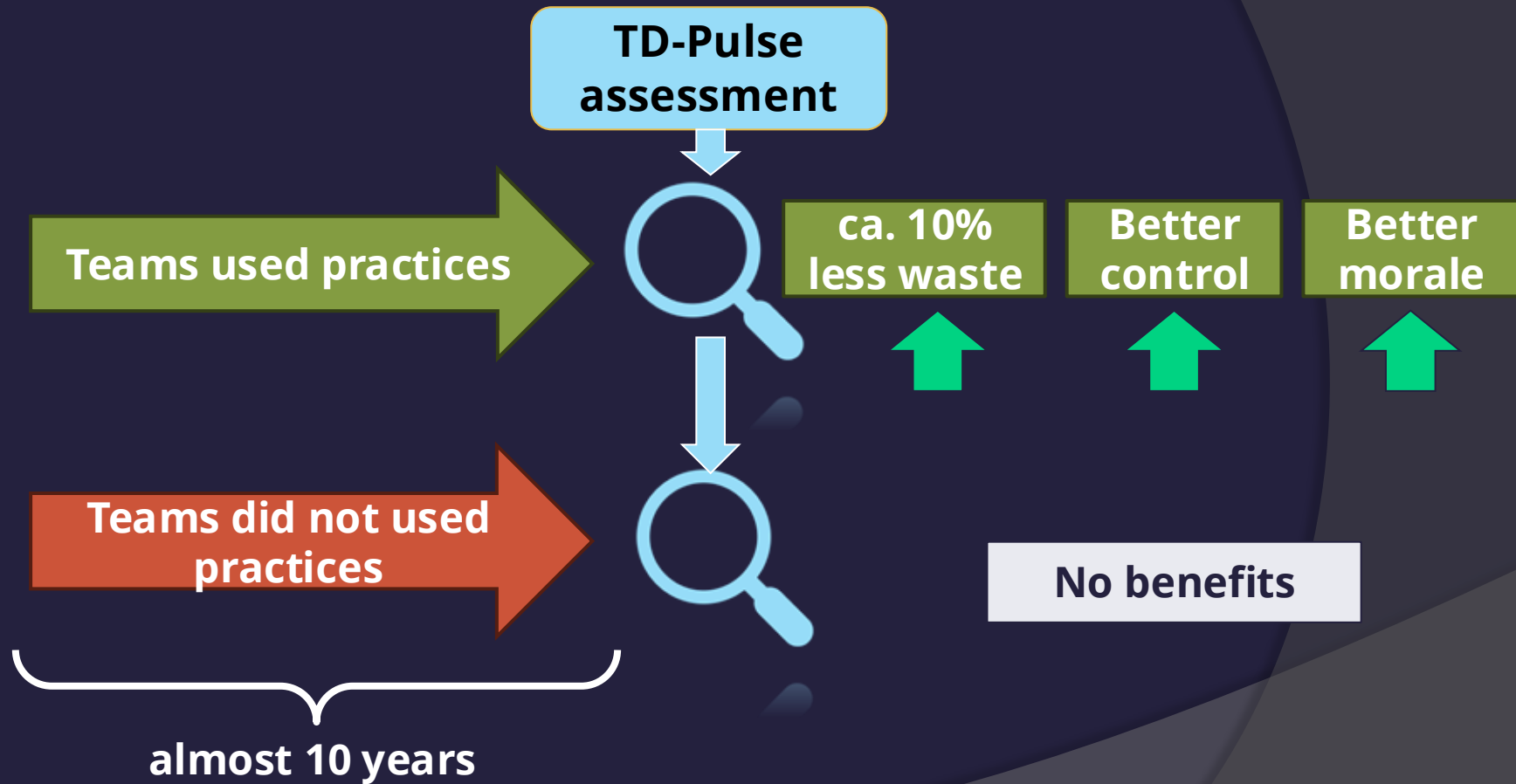
TDMA associates with more technical roles using tools to identify, prioritize and visualize TD



Other variables do not seem to be affected by TDMA presence or not

None of the variables are affected negatively by the presence of TDMA

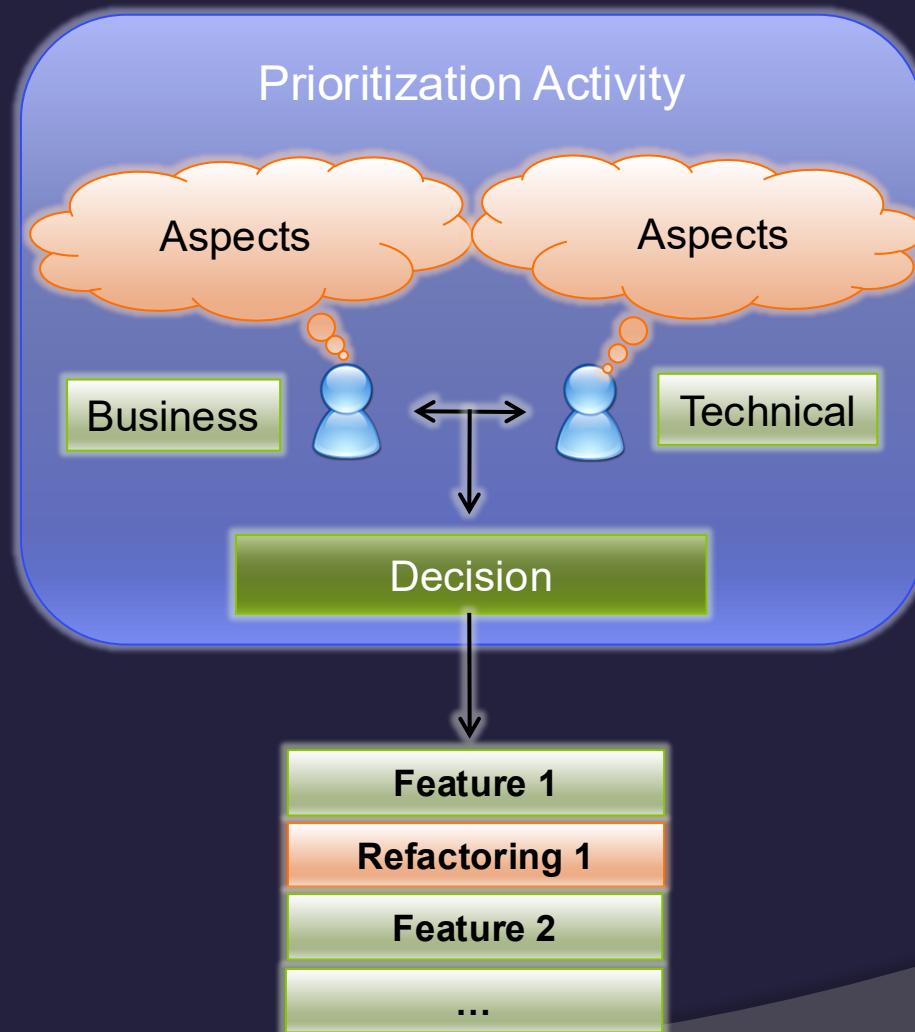
Summary of results



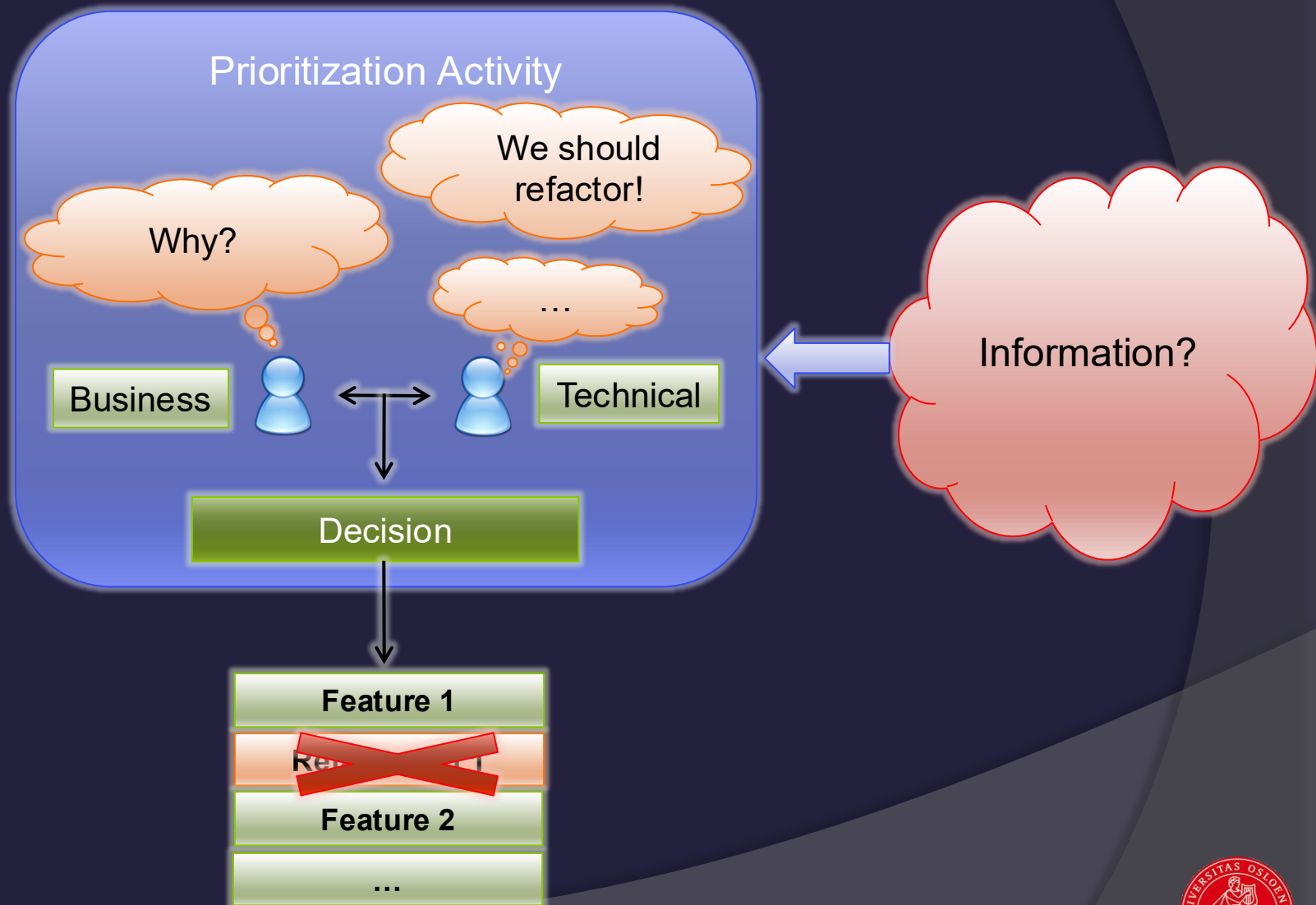


Prioritizing Technical Debt

Background: Prioritization of Features and Refactorings (of Technical Debt)



Problem 1: Refactoring is not prioritized



Problem 2: What to refactor? Can we wait?



Refactoring

Refactoring

Refactoring

Refactoring

...

...

...

...

Refactoring



Should we prioritize refactoring? In theory, it's simple... *

Principal (Cost of Refactoring)

Total Interest (Negative Impact)

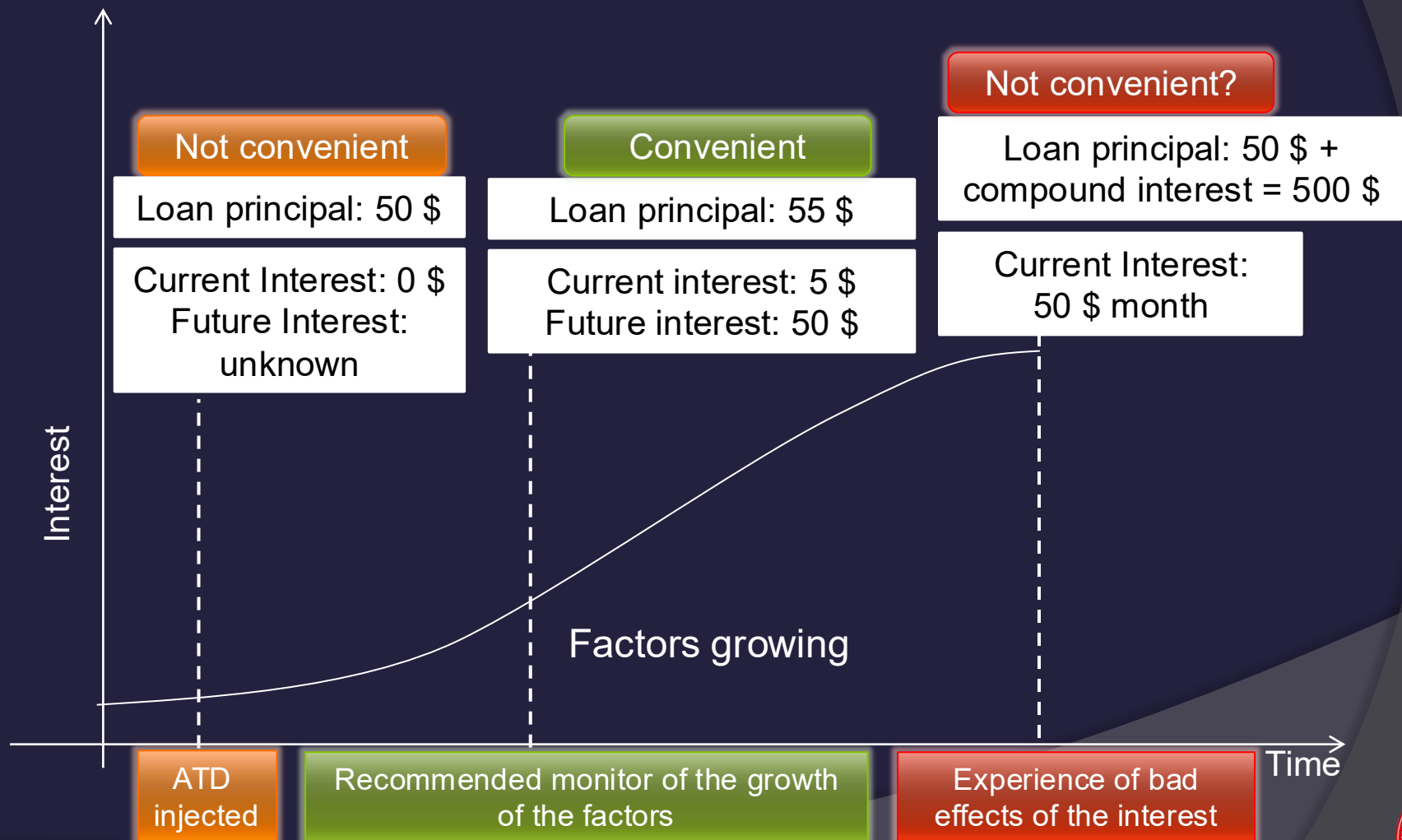
$\ll 1$

When the cost of refactoring is (much) greater than the interest, it's not convenient to refactor

* Antonio Martini and Jan Bosch. 2016. An empirically developed method to aid decisions on architectural technical debt refactoring: AnaConDebt. In Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)



What's the problem with refactoring TD?



Method: AnaConDebt* – will the interest grow?

now

future

time →

Current Analysis

Current Principal

Current Interest

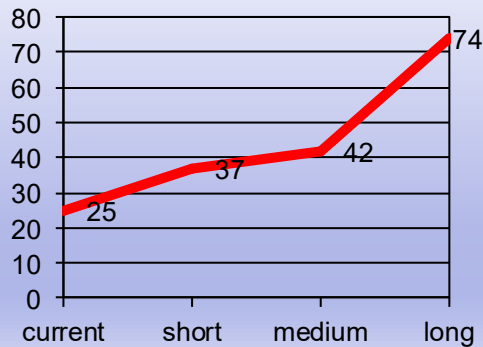
Evolution

Future Analysis

Future Principal

Future Interest

Example: Additional cost for refactoring increments



Internal Propagation Factors

External Propagation Factors

Internal Impact

External Impact

Indicators

If Decision

When Decision

Gathered and refined through several evaluative iterations

Technical debt

- ⦿ The interface for a platform is complex and applications using it report problems
 - Many bugs
 - Slow feature release
 - Quality issues
 - Many users affected – reputation
- ⦿ Cost of refactoring
 - Source component
 - Depending components
 - Study of solution
 - Add and replace tests
- ⦿ Should we refactor?



Results: *if* decision - key factors

Depending components

Solution Analysis

Learning new solution

Complexity

Tests

Source

Documentation

Principal (Cost of Refactoring)

Total Interest (Impact)

Slow Bug fixes

Other costs

Slow feature development

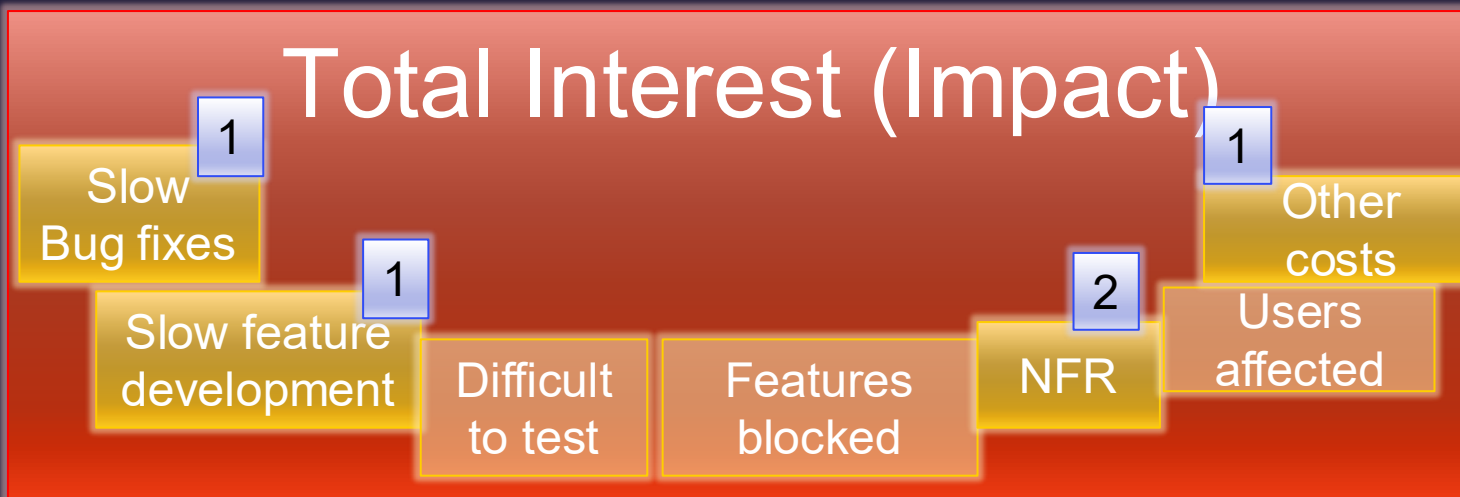
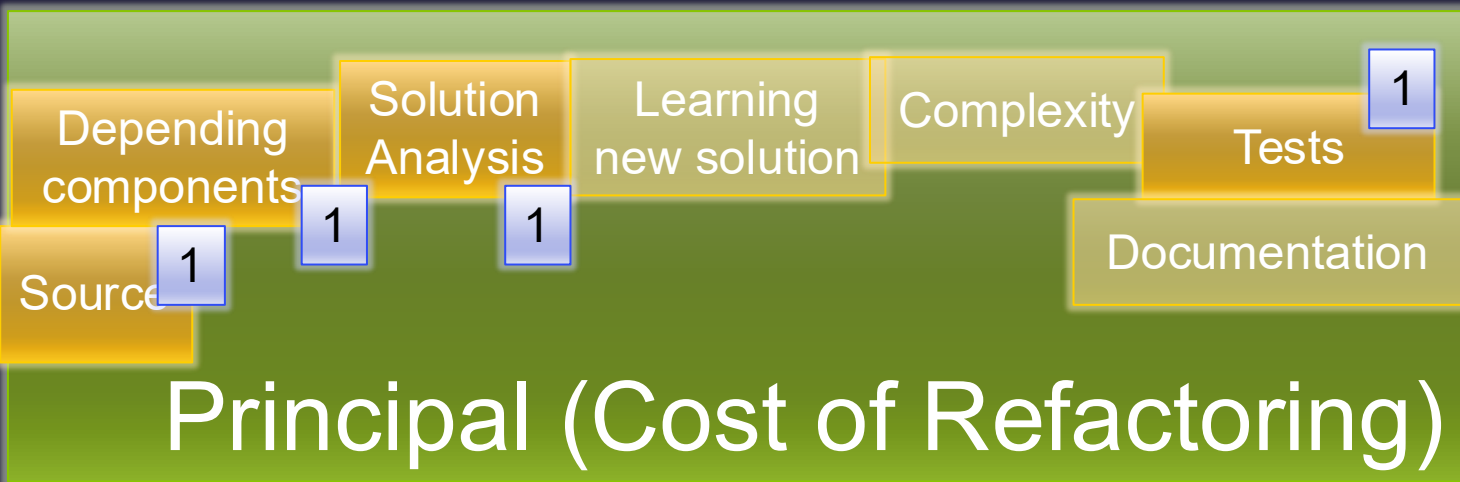
Difficult to test

Features blocked

NFR

Users affected

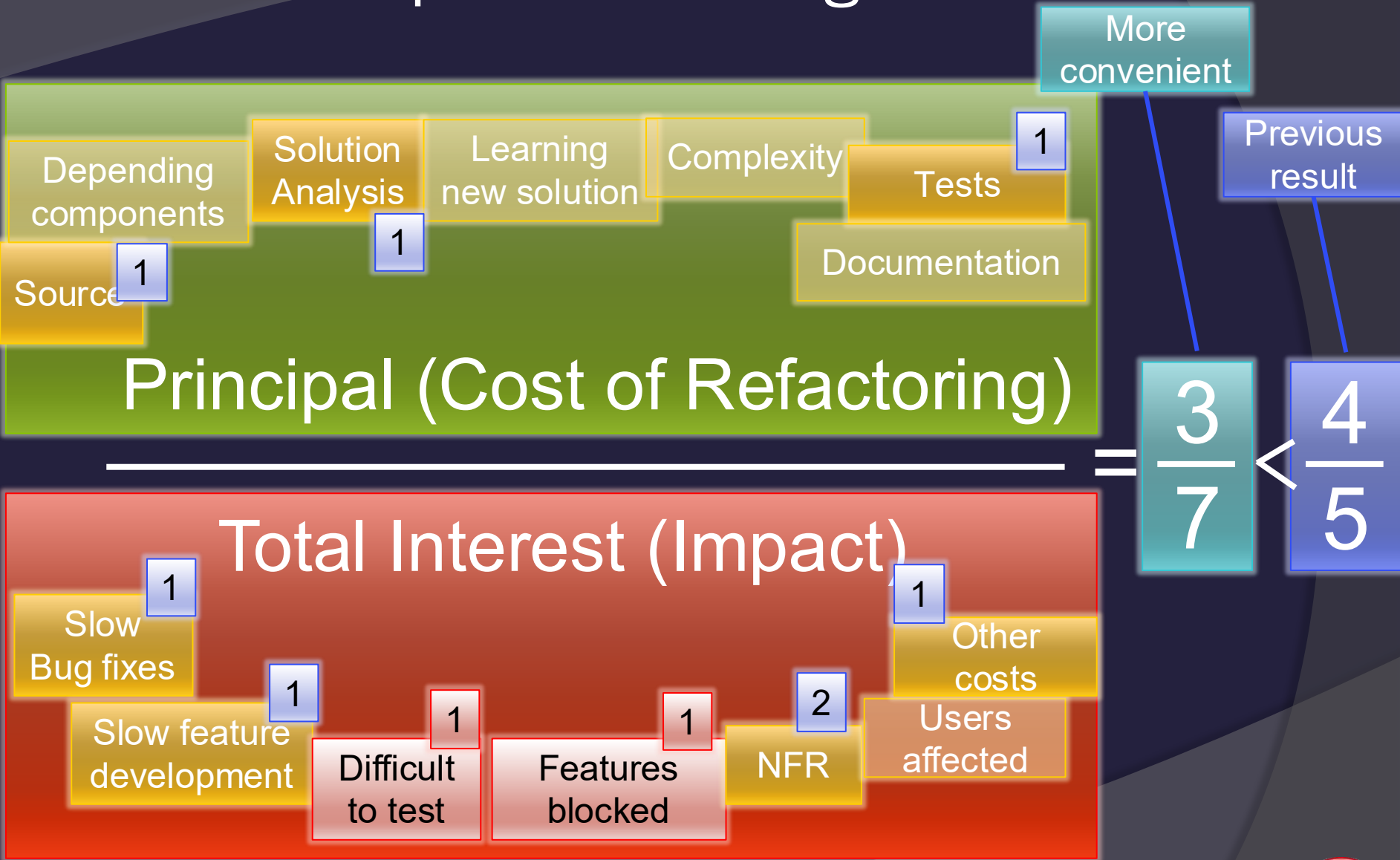
Results: *if* decision - weights for the factors



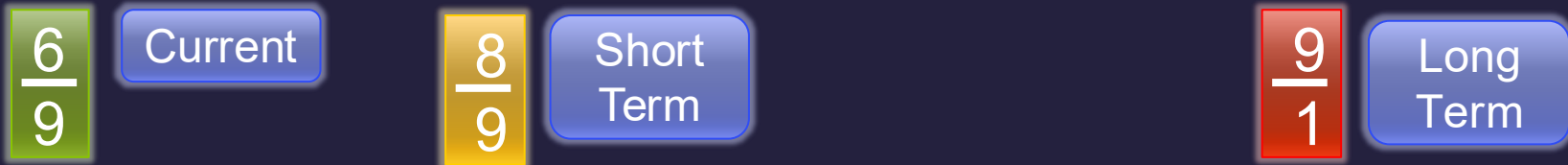
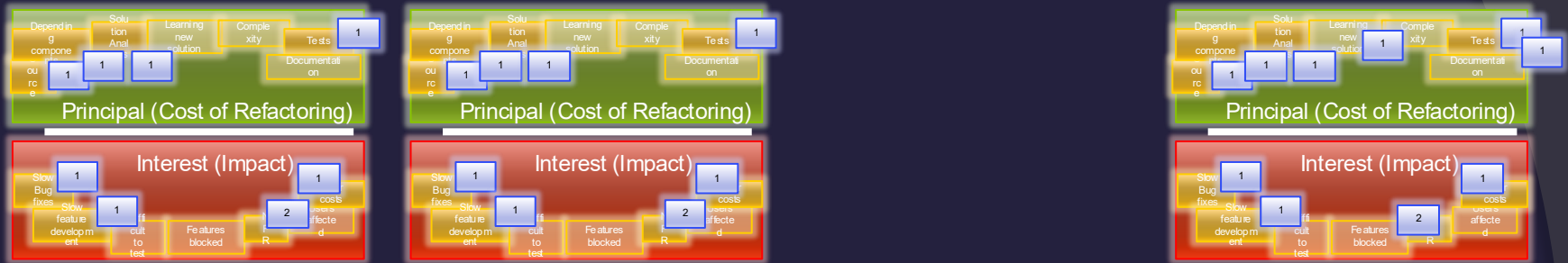
$$= \frac{4}{5} < 1$$



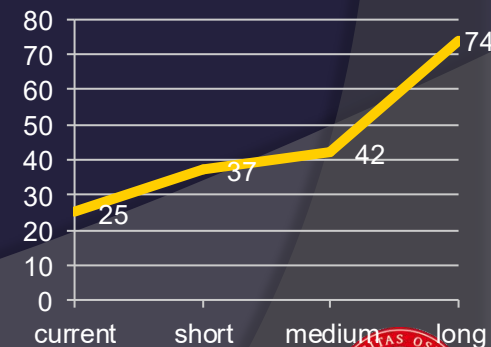
Results: comparison among ATD items



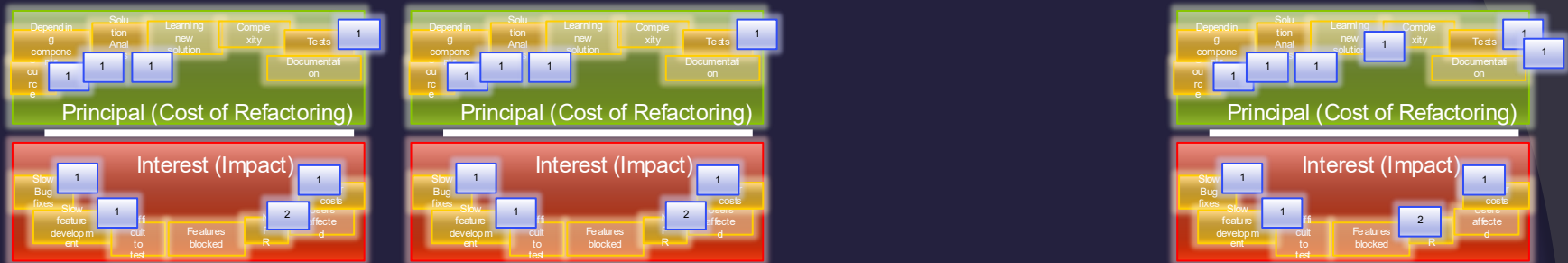
Results: *when* decision - comparison over time



Estimated based on current information and propagation factors (e.g. roadmap, users, etc.)



Results: *when* decision – iterative comparison



Iteration 2...



Take aways

Technical debt is a “sticky”, expensive and “creeping” problem



~ 30% waste

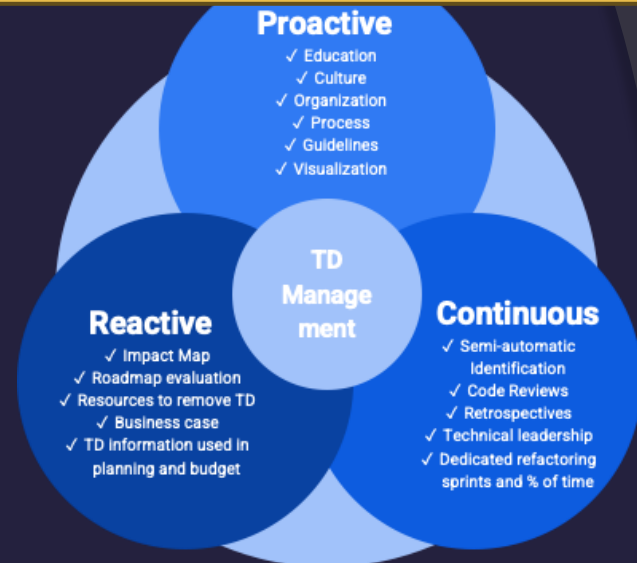
We need to prioritize:
Several factors over time

Principal (Cost of Refactoring)

Total Interest (Negative Impact)

$\ll 1$

Technical Debt is not (only) in the code
The causes are on a different level



Several practices can be used

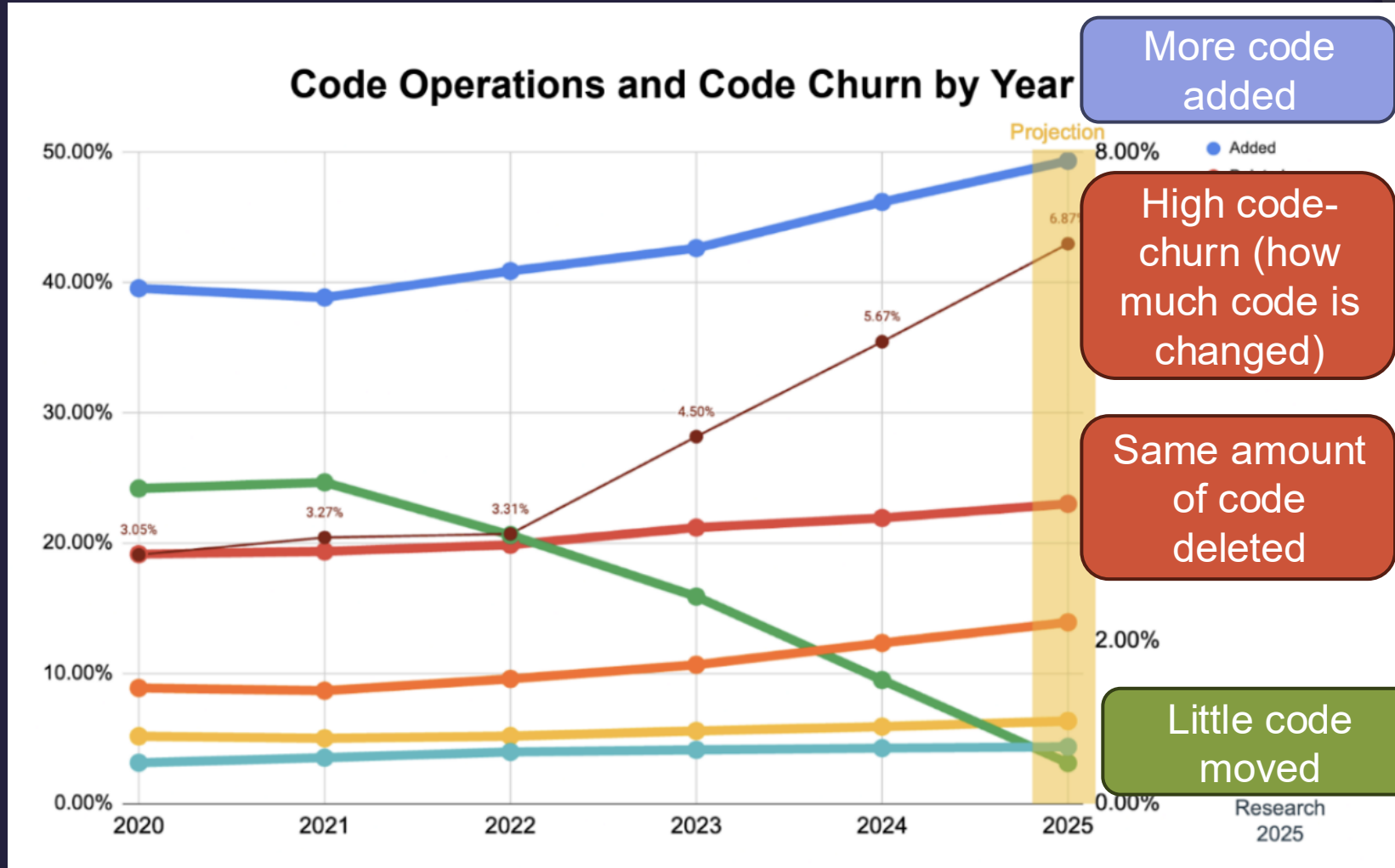
They work:

- -10% waste
- Better control
- Better morale

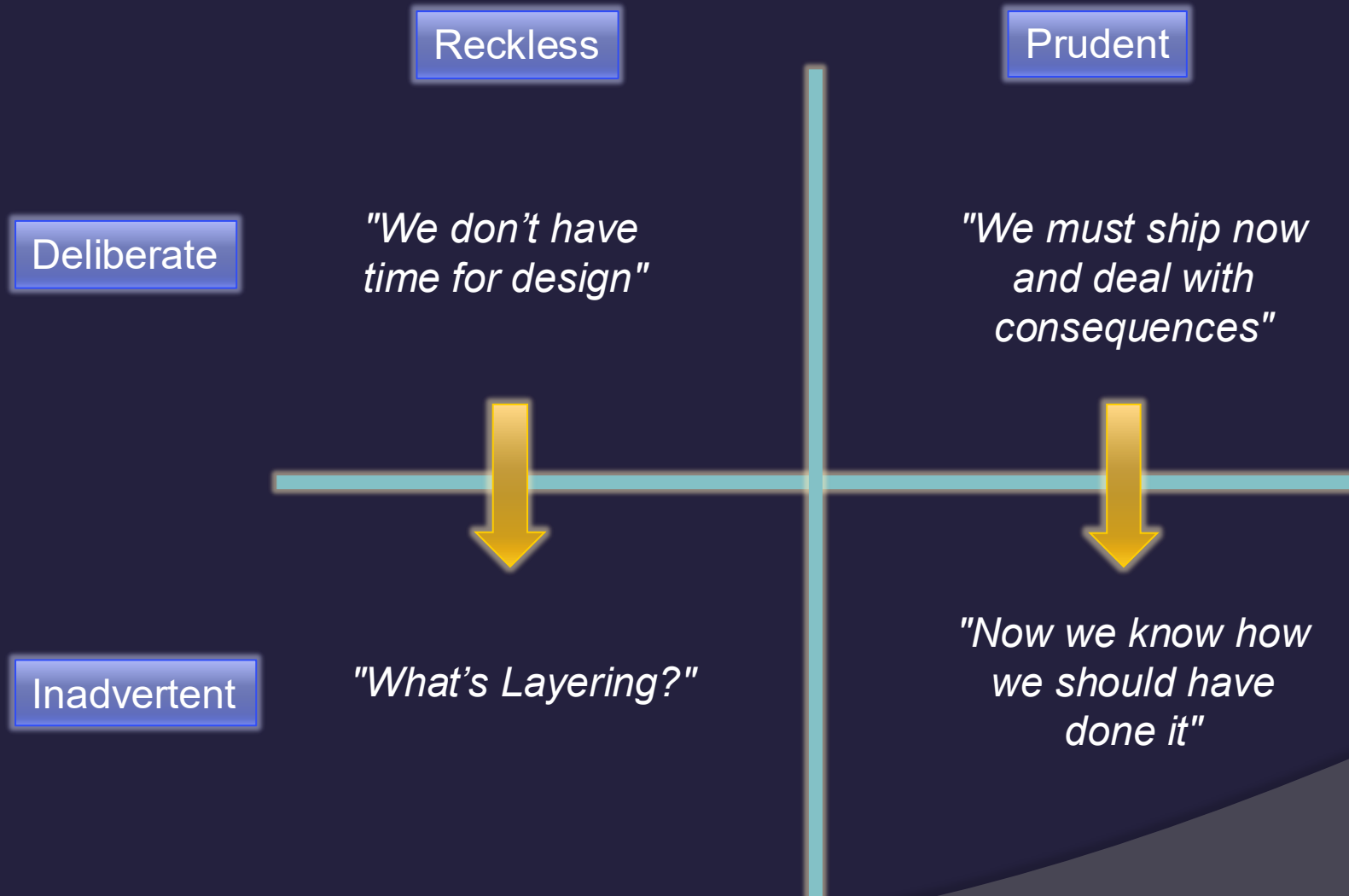
Technical Debt and AI



Does the use of AI generate more or less technical debt?



Technical Debt and AI

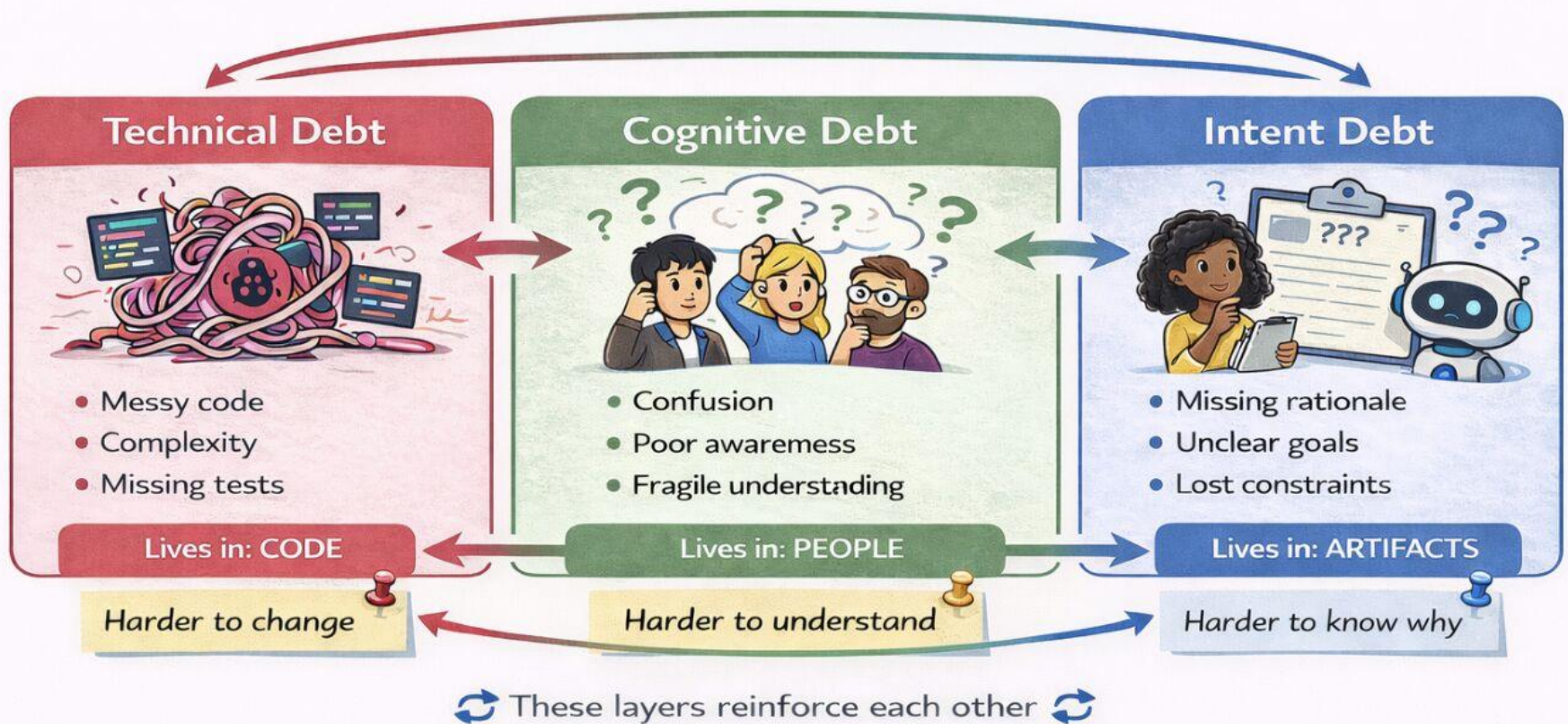


* martinfowler.com/bliki/TechnicalDebtQuadrant.html

The triple debt*

Triple Debt Model

Three Layers of Software Health



Other slides



Interest Map

Category	Factors
business	competitive advantage
	lead time
	attractiveness for the market
	penalties
	feature usage
	business value
	ROI (calculated per bug)

TD Impact Map *

What are the risks of TD?

other factors	contagious debt
	existence of TD solution (alternative)
	spread of impact in the system
	number of users affected
	frequency of negative impact
	kind of smell
	history of the system
	compromise architecture
	future cost
	user perception
not specified	risk
	interest likelihood
	interest
	severity

customer	satisfaction
	long-term satisfaction
	specific customer value
	customer expectations
evolution	customer effect
	time of impact on evolution (short or long-term)
	risk of critical impact on evolution (possible crisis)
	impact on other features
maintenance	impact on upcoming features
	modifiability
	number of bugs
	maintenance cost
productivity	% wasted time (effort)
	number of developers working on TD
	wasted development hours
	generic effort
project factors	coding output/effort
	availability of resources
	project size and complexity
	postponement of bugs
quality debt	# of issues or their co-occurrence
social factors	developers' morale
	social debt
	positive impact of TD
	team culture
system qualities	robustness
	performance efficiency
	security
	transferability
	scalability
	generic qualities

* Valentina Lenarduzzi, Terese Besker, Davide Taibi, Antonio Martini, Francesca Arcelli Fontana, *A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools*, Journal of Systems and Software.

