Antonio Martini

Associate Professor in Software Engineering
Oslo University

Principal, Strategic Researcher
CA Technologies, Barcelona, Spain

Course IN2001
2018-02-13

ARCHITECTURE IN ANDROID AND TECHNICAL DEBT

Who is Antonio Martini?

- Italian
  - No kebab pizza! ☺
  - 7 years in Scandinavia– survived many winters!
- Previously
  - Worked as a Software Developer
  - PhD in Software Engineering
  - Postdoc at Chalmers, Gothenburg
  - Independent Consultant
    - project with Ericsson
    - project with Volvo Group
- Currently:
  - Principal, Strategic Researcher at CA Technologies
  - Associate Professor at Oslo U.
  - Independent Consultant
- Hobbies
  - Board games, strategy computer games, pool, etc.
  - Football, volleyball, beach volley, fencing
  - Piano, Drumset, etc.
  - Travel!
  - …and no time for them! ☺

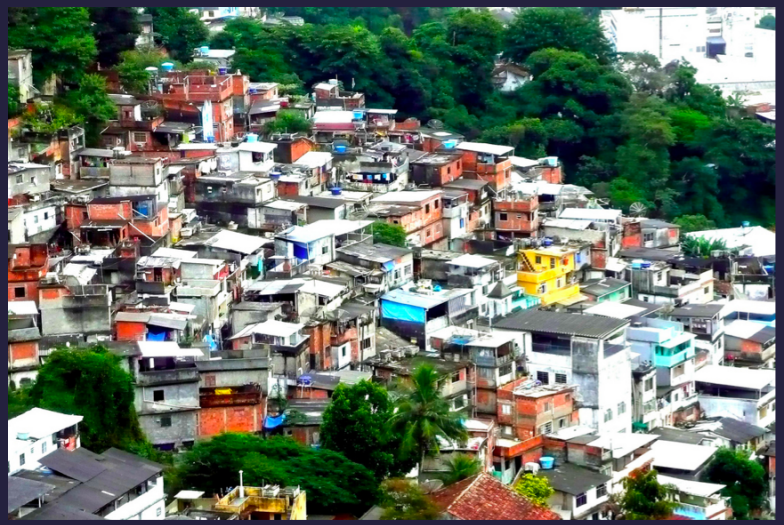
ca technologies

Worked with and for several companies

What is Software Architecture?

ca technologies

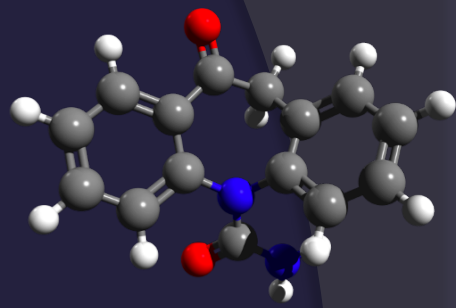
What's the difference?



Where can you run public transport efficiently?
Which city is easier to grow?
Where do you have a good emergency system?
…

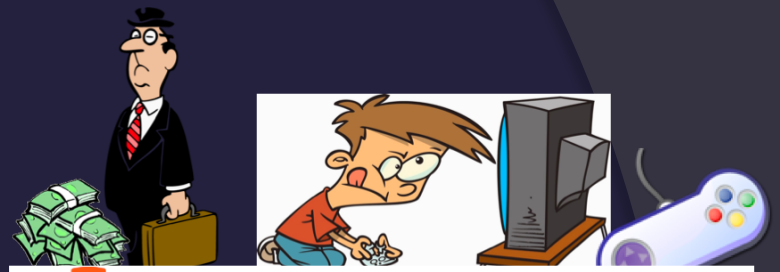
Software architecture is...

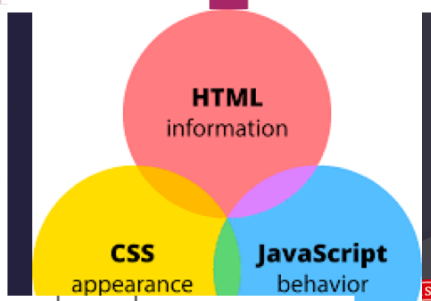
- All of the followings:
  - Overall system **structure**
  - A set of architectural **design decisions**
  - Things that people perceive as **hard to change**
  - The "**important** stuff" – whatever that is

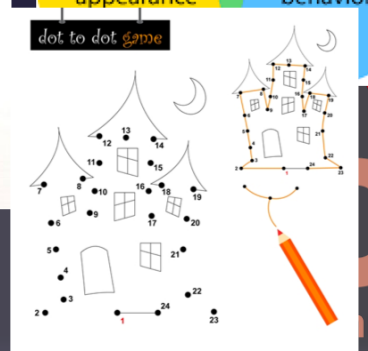


Software Architecture characteristics

- Multitude of stakeholders
- Quality driven (tradeoff)
- Separation of concerns
- Recurring styles (patterns)
- Conceptual integrity (vision)







Why software architecture?

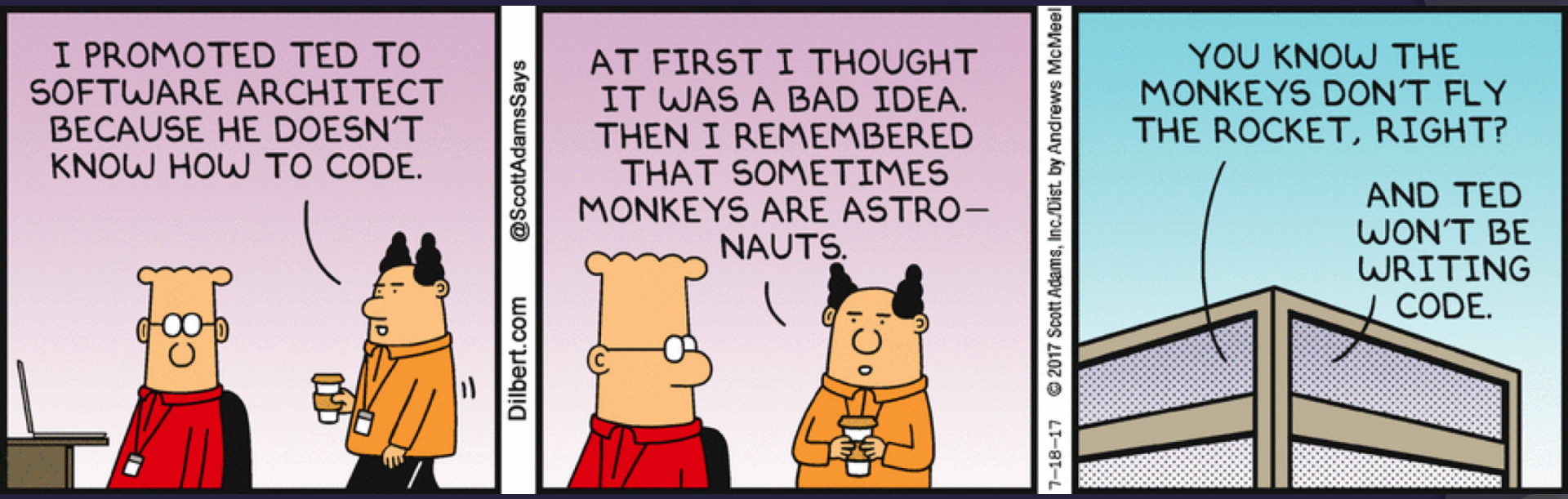
- To get a grasp of a complex system
- Facilitates the communication among the stakeholders about their needs
- Support decisions about future development and maintenance
  - Reuse
  - Budget
- Analysis of the product before it's built
  - Cost reduction
  - Risk reduction

Size does(n't) matter

- All products HAVE an architecture
  - It can be bad
  - It can be good
- In all projects we SHOULD think about architecture
  - Less in small projects
  - More in large projects
- Thinking about the architecture is a necessary process

ca technologies

Don’t undervalue architecture...



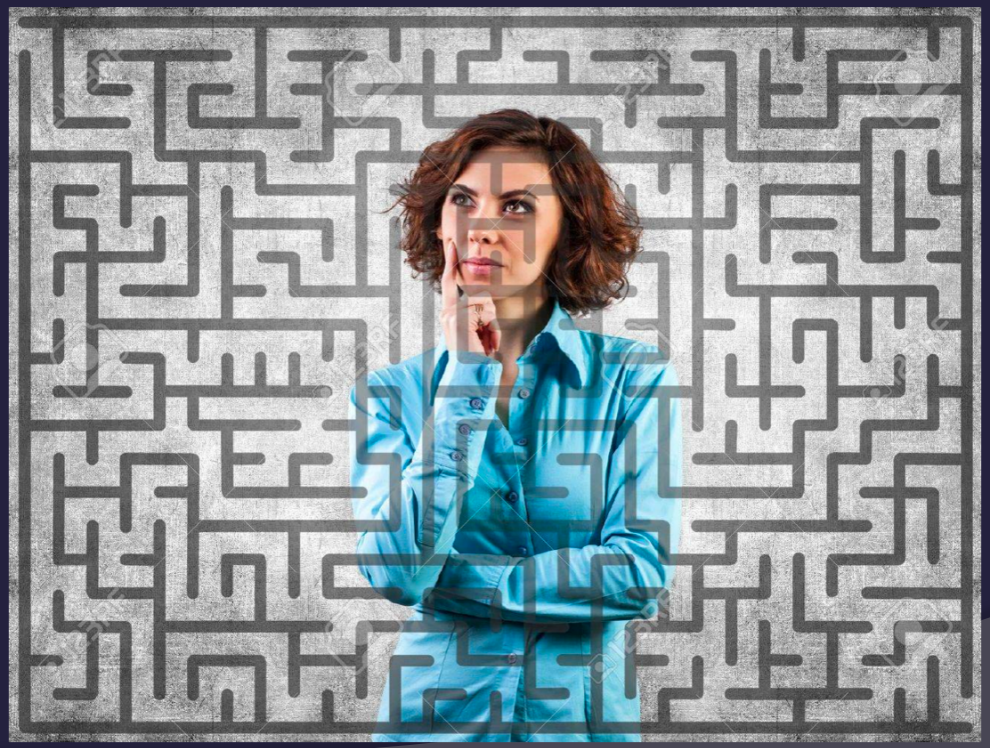
How to think about Architecture

Antonio Martini - Associate Professor in Software Engineering

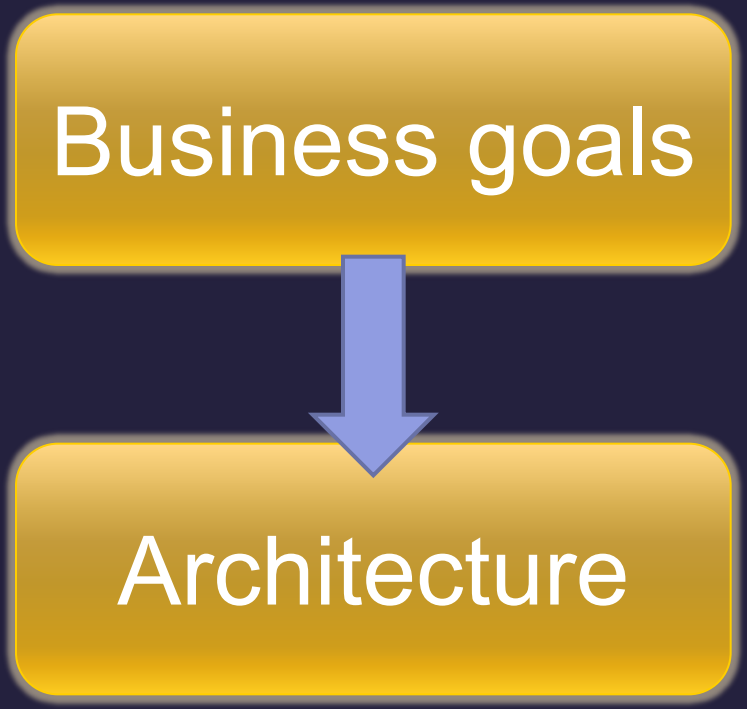
ca technologies

How to choose an architecture

- It can be quite difficult
- Where do we start?



Business drives architecture



A process to think about architecture

Who?

What do they need?

What should the system do?

What qualities are important?

What should we focus on?

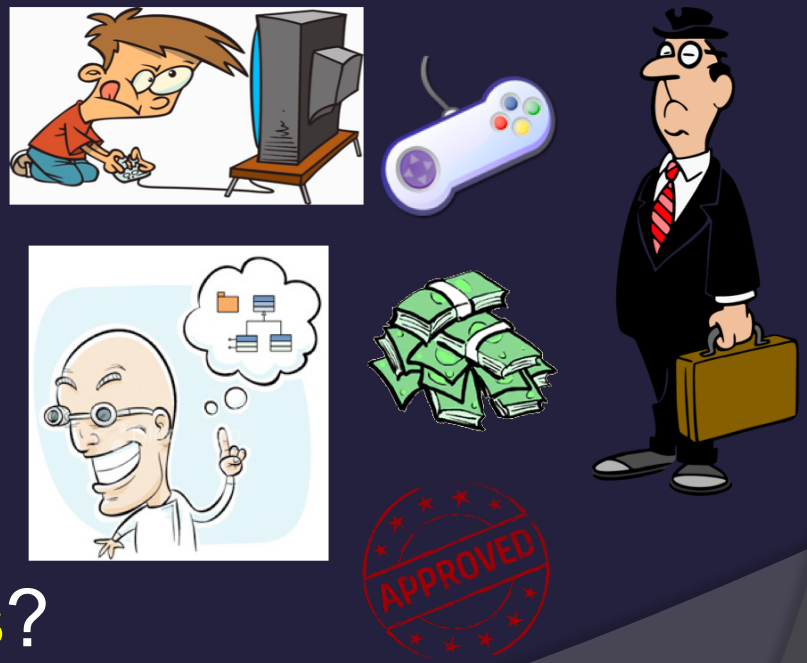
How should we implement it?

Stakeholders analysis (1)

- You might need to accommodate several stakeholders
- Stakeholder: "*an individual, group, or organization, who may affect, be affected by, or perceive itself to be affected by a decision, activity, or outcome of a project*"
- Who are the main stakeholders of your app?

Stakeholders analysis (2)

- Let's consider the three stakeholders below:
  - User of the app
  - Sales
  - Engineers



- What are their needs?
  - Write down 2 important needs for each stakeholder

Needs examples

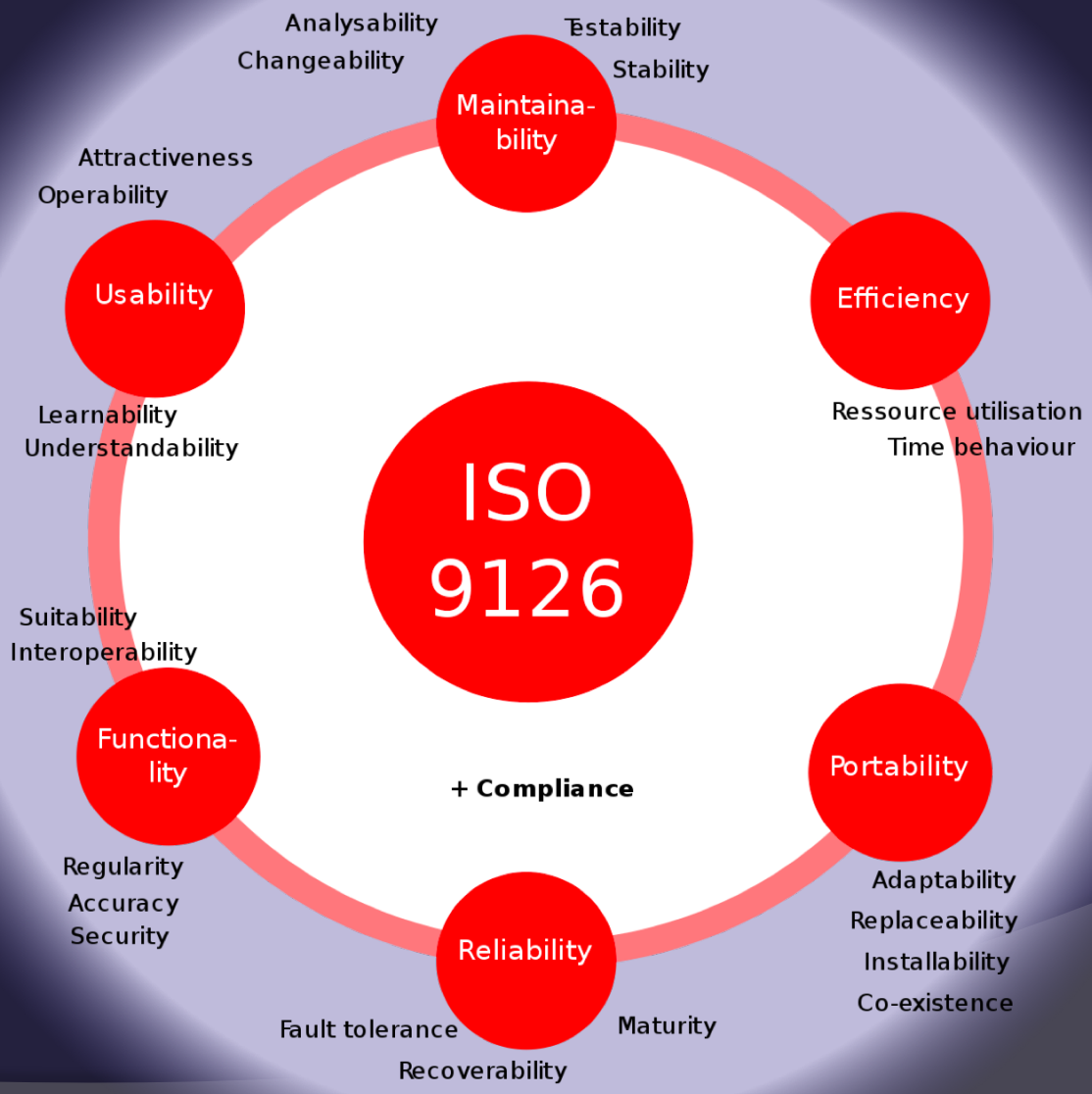
- Sales' needs:
  - *"we need to deliver the app fast"*
  - *"we need the app to be available for both Android and iOS"*

- Users' needs
  - *"we want to have an experience without bugs"*
  - *"we want to get the information quickly"*

- Engineers' needs
  - *"We need to test the app easily"*
  - *"We need to be able to add features quickly after the first release"*

- Example of a need that we don't have: *Security*

System Qualities



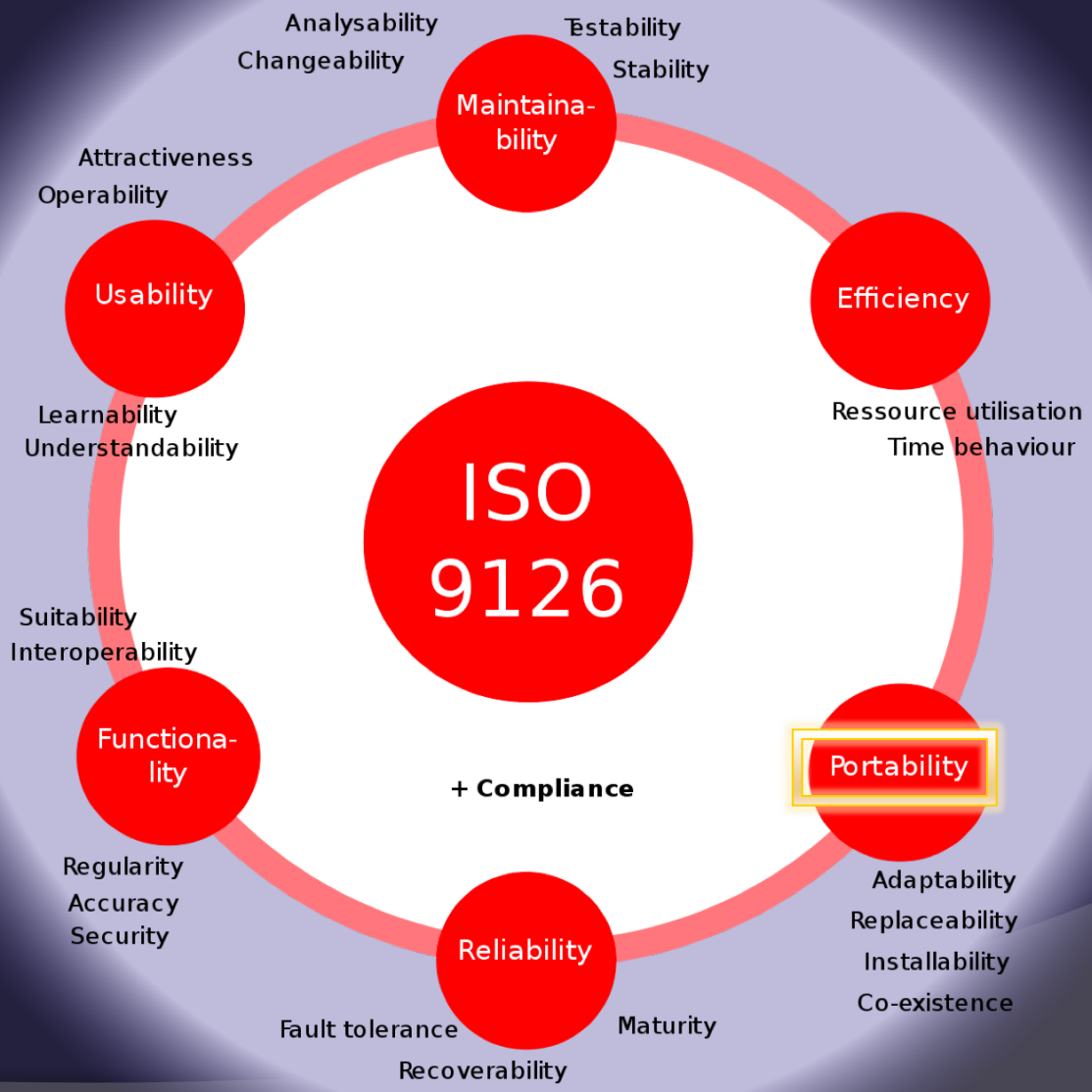
https://www.iso.org/standard/22749.html

From needs to qualities - sales

- Sales' needs:
  1. *"we need to deliver the app fast"*
  2. *"we need the app to be available for both Android and iOS"*

- Qualities?
  1. *No quality – Budget constraint*
  2. *Portability*

System Qualities - Sales

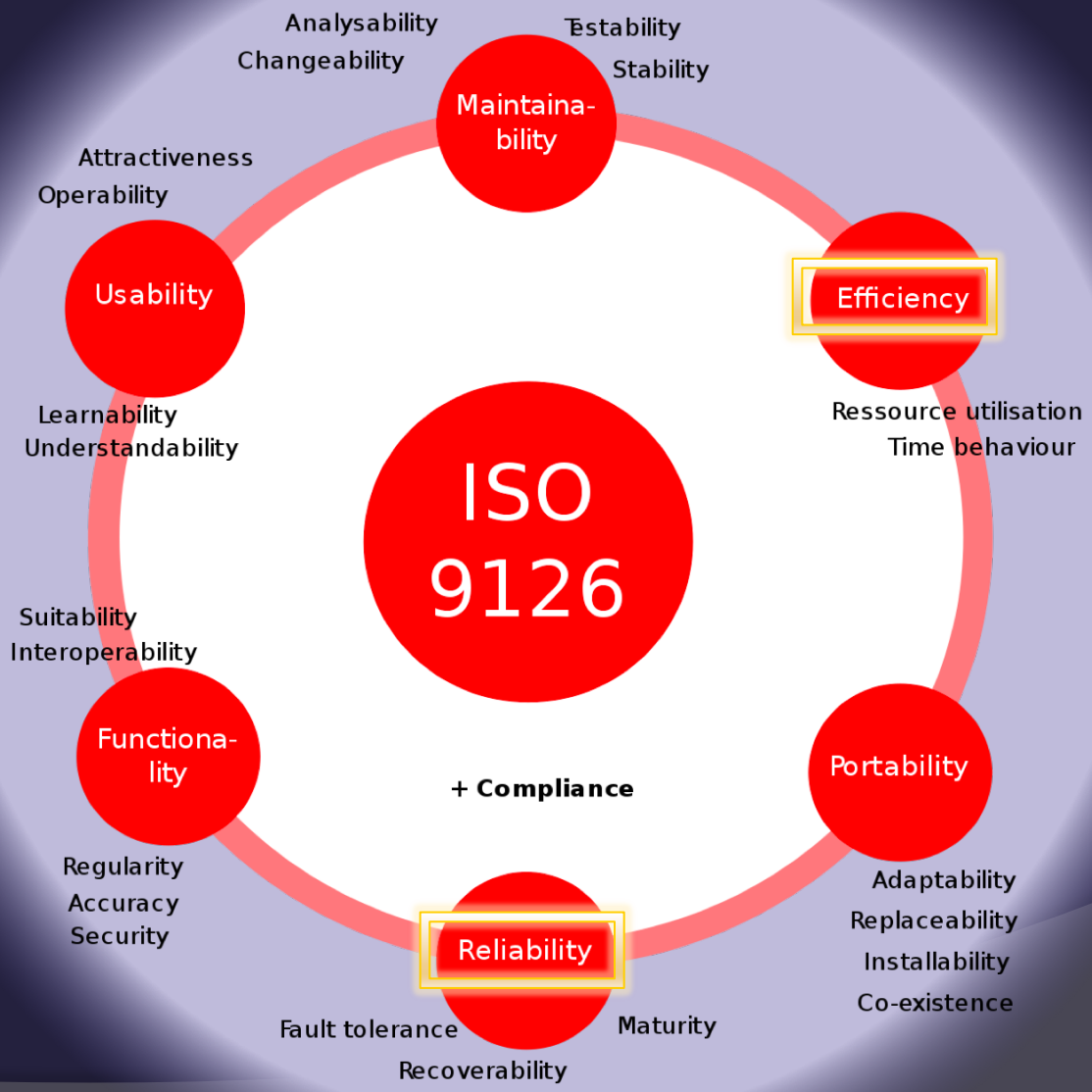


From needs to qualities - users

- Users' needs
  1. *"we want to have an experience without bugs"*
  2. *"we want to get the information quickly"*

- Qualities?
  1. *Reliability*
  2. *Efficiency (Performance)*

System Qualities – Users



technologies

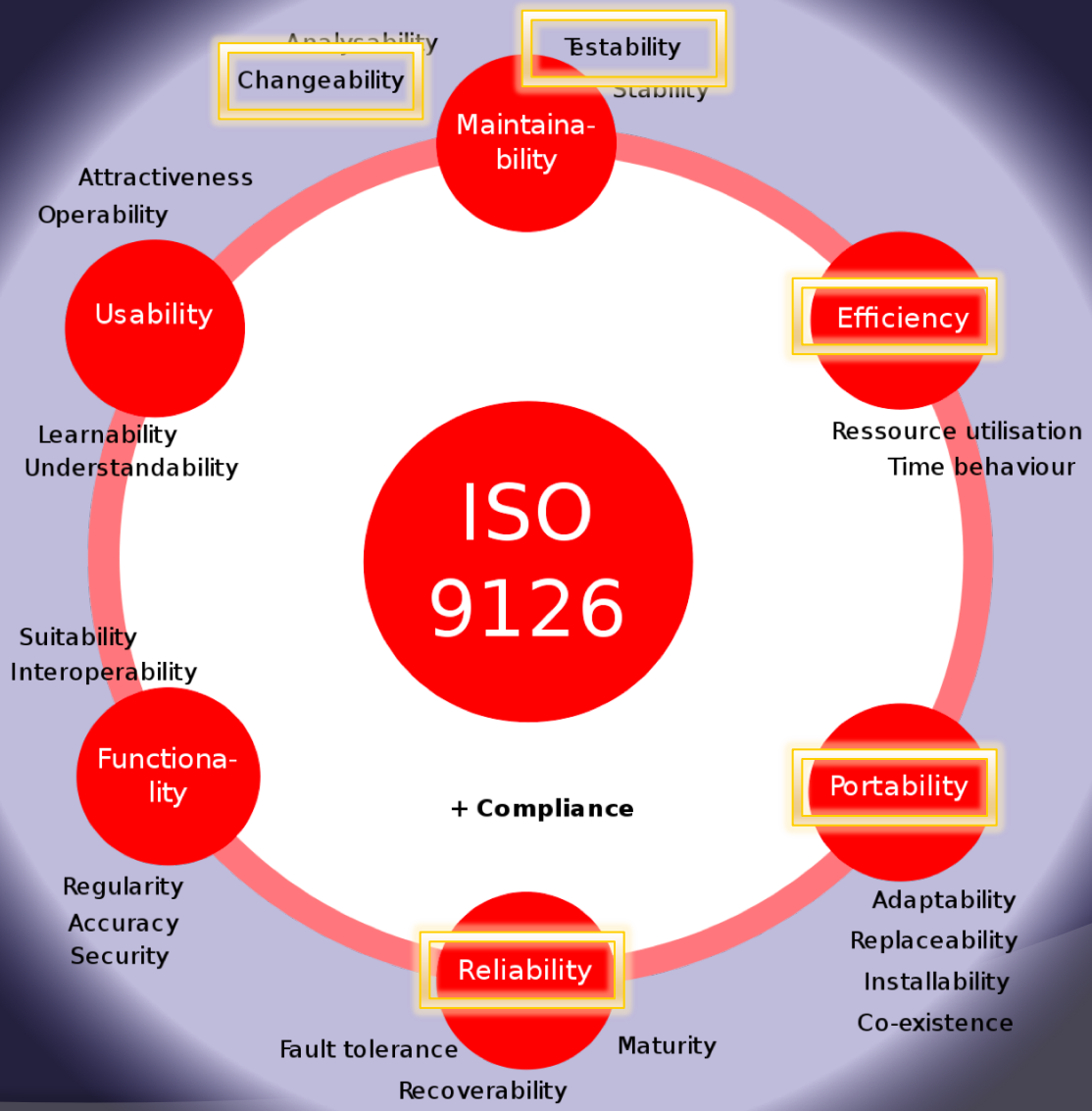
From needs to qualities - engineers

- Engineers' needs
  1. *"We need to test the app easily"*
  2. *"We need to be able to add features quickly after the first release"*

- Qualities?
  1. *Testability – Mantainability*
  2. *Changeability – Maintainability*

System Qualities - Engineers

System Qualities – All stakeholders



Can we afford to say yes to everyone?

- Are there some conflicts?

- Example?
- Sales' needs
  1. *"we need to deliver the app fast"*
  2. *"we need the app to be available for both Android and iOS"*
- Or else:
  1. *Budget constraint*
  2. *Portability*

- Can we achieve both?

ca technologies

Can we say yes to both needs?

- We investigate further the details.
  We discover that:
  - Sales want to deliver in 3 months
  - To make the app portable both for Android and iOS, we need to:
    - Use special libraries
    - Learn more skills
    - Test in more environments
  - Conclusion: it takes 5 months
- The answer is NO. What do we do?
  - We ask the stakeholders to prioritize the needs
  - We reach a tradeoff

ca technologies

Tradeoff(s)

- We generate solutions and scenarios
  1. Solution 1:
     - We take 5 months to make the product portable
     - We deliver in 5 months
  2. Solution 2:
     - We deliver in 3 months
     - We make the app portable later on

- Which one do we choose? Why?

Cost/Benefit and risk analysis

- Which solution is best?
  - Solution 1
    - Waiting 2 months more (5-3) costs us several customers
      - Risk: competitor app might “steal” our customers
      - Risk: if another app steals our customers we don’t get visibility in media
  - Solution 2
    - It will cost more to deliver
      - We need to deliver the app in 3 months for Android
      - We will need to re-write it for both platforms
      - Total: 3 months + 4 to rewrite = 7 months
    - But we reach the customers of one platform soon
      - We gain visibility

Scenarios and analysis

	Benefit: Users short-term	Benefit: User long-term	Cost	Total
Solution 1	-- (vs competitor)	++ (both platforms) - (lack of visibility)	+ (cheaper in total)	0
Solution 2	++ (vs competitor)	+ (visibility) - (no users in one platform)	- (rewrite)	+1

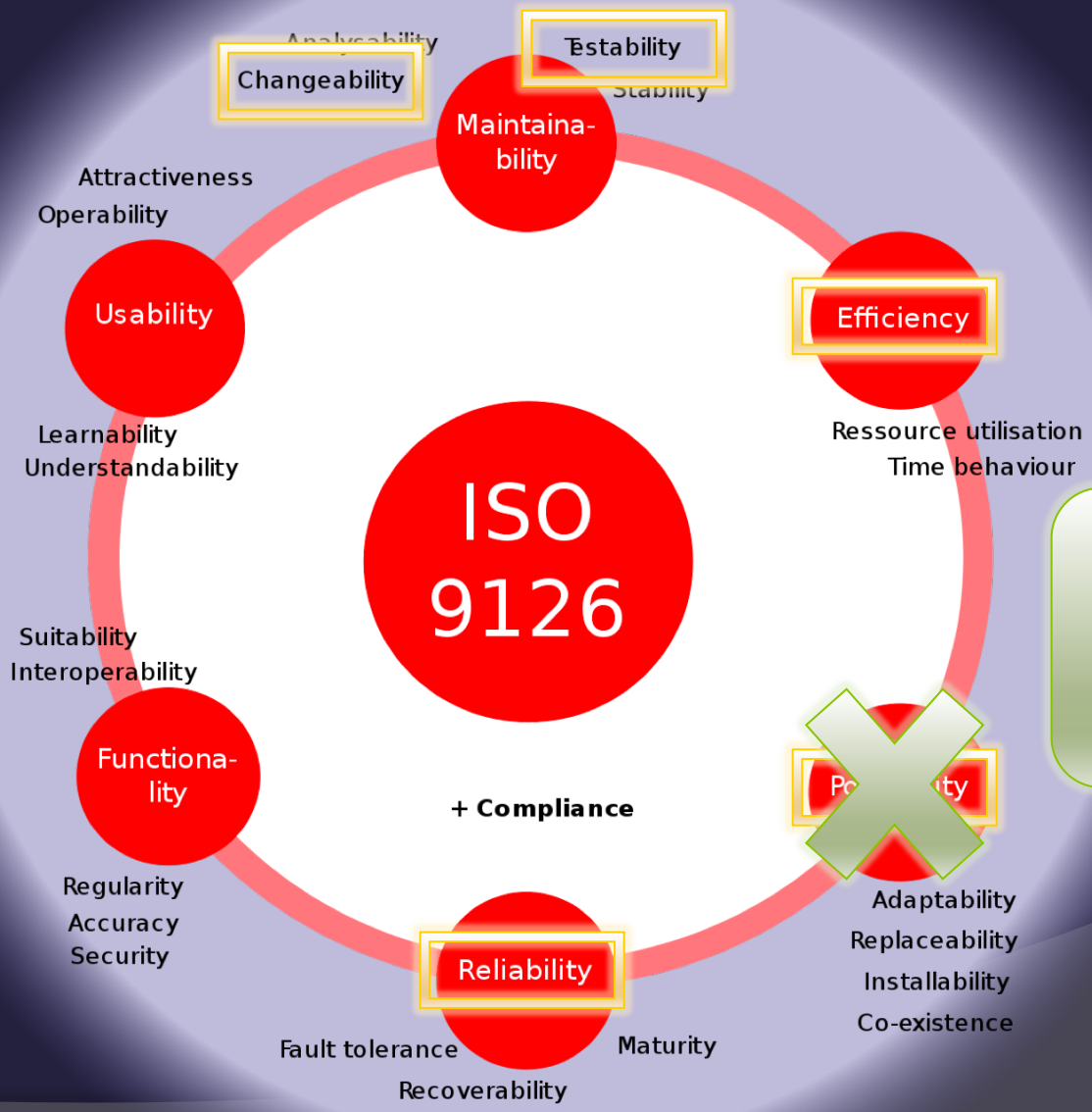
Tradeoff(s) example

- We generated solutions and scenarios
  1. Solution 1:
     - We take 5 months to make the product portable
     - We deliver in 5 months
  2. Solution 2:
     - We deliver in 3 months
     - We make the app portable later on

- Which one do we choose?
  - We choose **Solution 2**
    - We deliver the app in 3 months
    - We skip portability for now
- Why?
  - Because it's better according to the cost/benefit analysis

ca technologies

System Qualities – All stakeholders



Tradeoff: we need to postpone portability

technologies

Are there other conflicts?

- Sales' needs:
  - *"we need to deliver the app fast"*
  - *"we need the app to be available for both Android and iOS"*

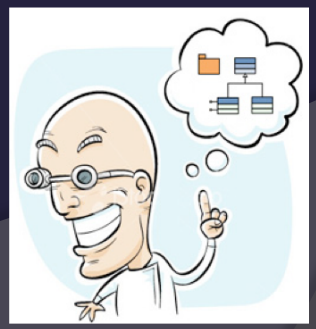
- Users' needs
  - *"we want to have an experience without bugs"*
  - *"we want to get the output quickly"*

- Engineers' needs
  - *"We need to test the app easily"*
  - *"We need to be able to add features quickly after the first release"*

Another (classical) conflict

- Sales
  - *"we need to deliver the app fast"*

- Engineers
  - *"We need to be able to add features quickly after the first release"*
  - Or else: Maintainability



Building a good solution

ca technologies

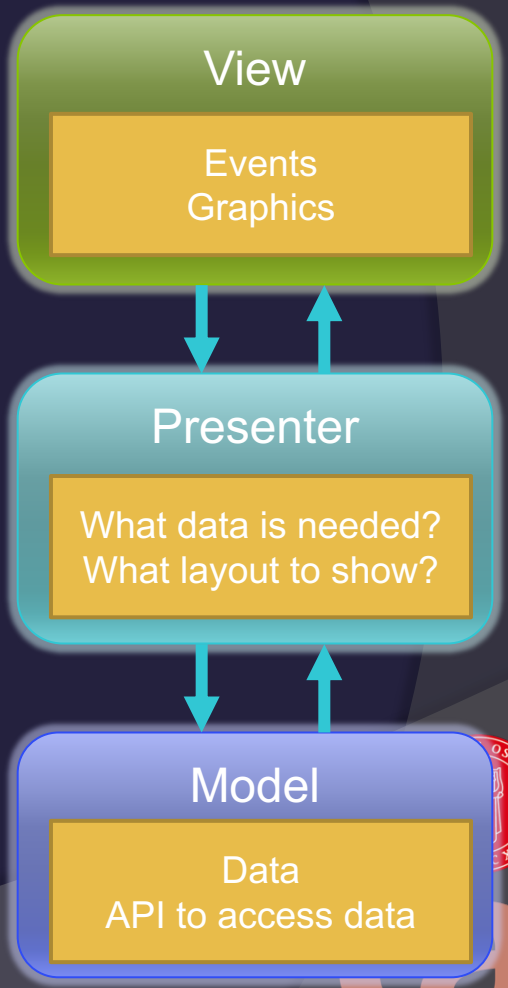
What does it mean Maintainable code?

- Changeable
- Testable
- …

- We need a good Separation of Concerns
  - Not all the code in one file
    - E.g. an Activity
  - Separate in different parts of the system (modules) what concerns different aspects of the system
  - E.g.
    - The data (database access)
    - The view (the user interface)

Good separation of concerns

- In Android the MVP architectural pattern is recommended
- We separate three layers:
  - **Model**:
    - Manage how all the data is stored and accessed
  - **View**:
    - Passively shows the data from the Model
    - Collects the events produced by the user
      - e.g. the “Tap”
  - **Presenter**:
    - interprets the user events and what data is needed
    - chooses the right way to show the results



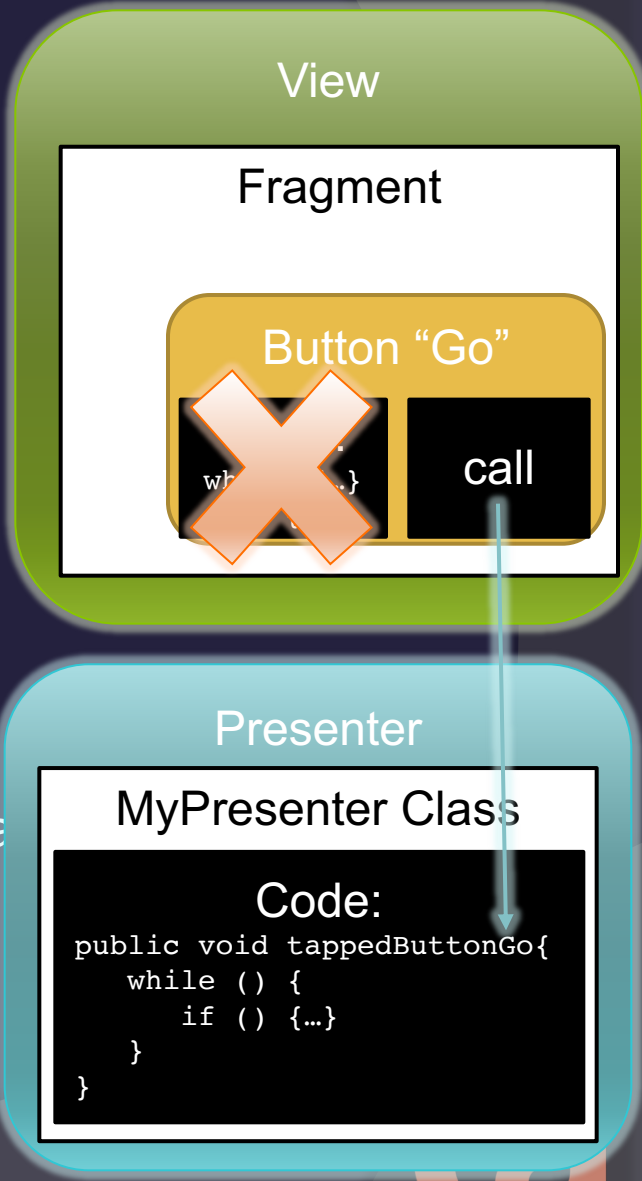
Architecture and MVP in Android

- Architecture guidelines in Android
  - https://developer.android.com/topic/libraries/architecture/guide.html#recommended_app_architecture
- Hands-on example of MVP on the web
  - https://medium.com/@cervonefrancesco/model-view-presenter-android-guidelines-94970b430ddf

ca technologies

A few guidelines (1)

1. Improve Testability
   - Write a “dumb View”
     - You don’t have to test a complex framework (Activity, Framework, …)
     - You only test your presenter (which you write yourself)
     - E.g.
       - When you write the code to execute for a button, do not write it in the Activity, but call a method in the Presenter



A few guidelines (2)

- Make Presenter Framework-Independent
  - Do not depend on Android classes when writing the Presenter
    - Much better to test!
    - Do not need an emulator

A few guidelines (3)

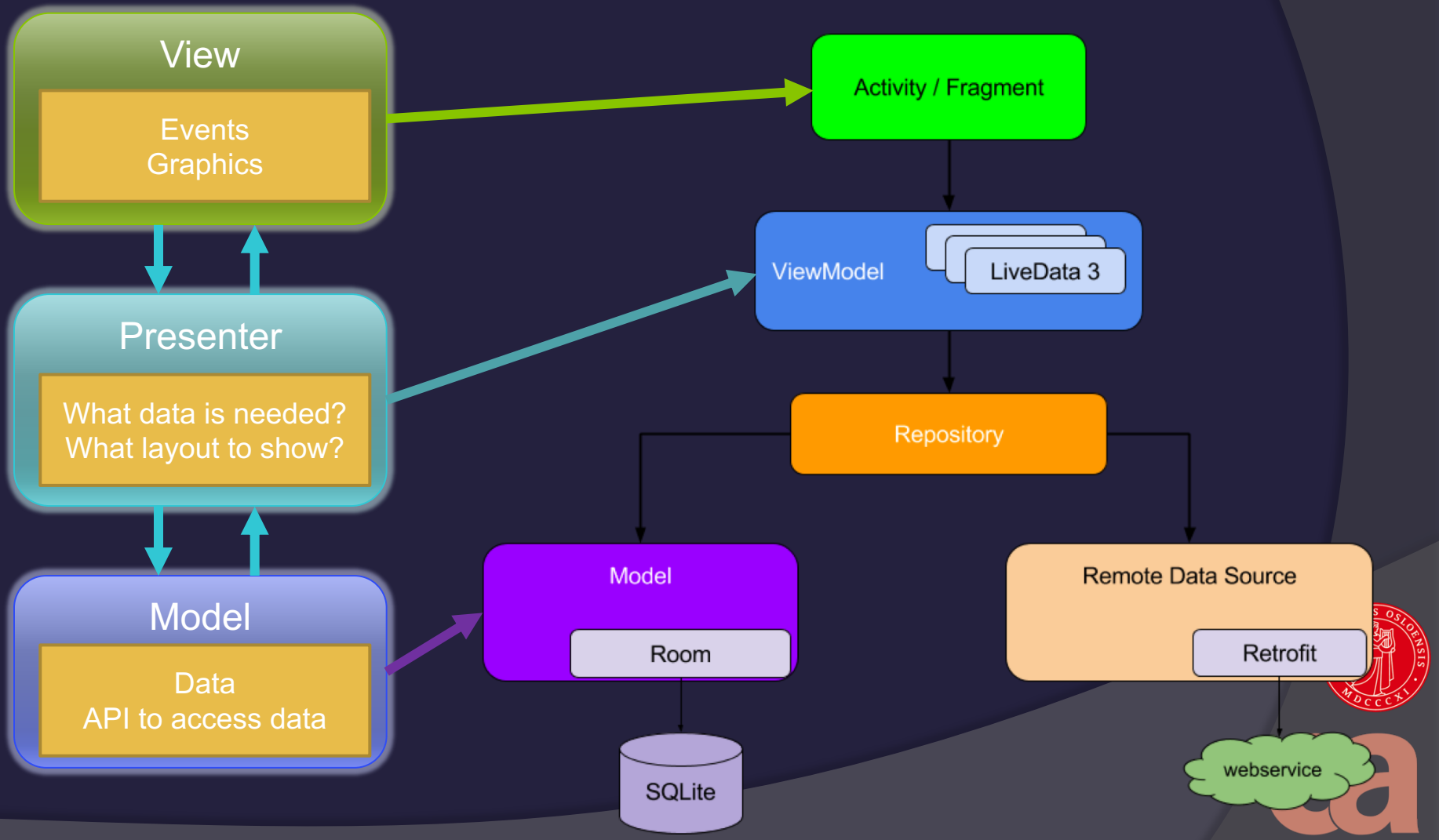
- Define naming conventions
  - Mainly 2 categories
    - Actions from the Presenter
      - load(), etc.
    - User Events interpreted by the Presenter
      - buttonGoPressed(), etc.

Other guidelines...

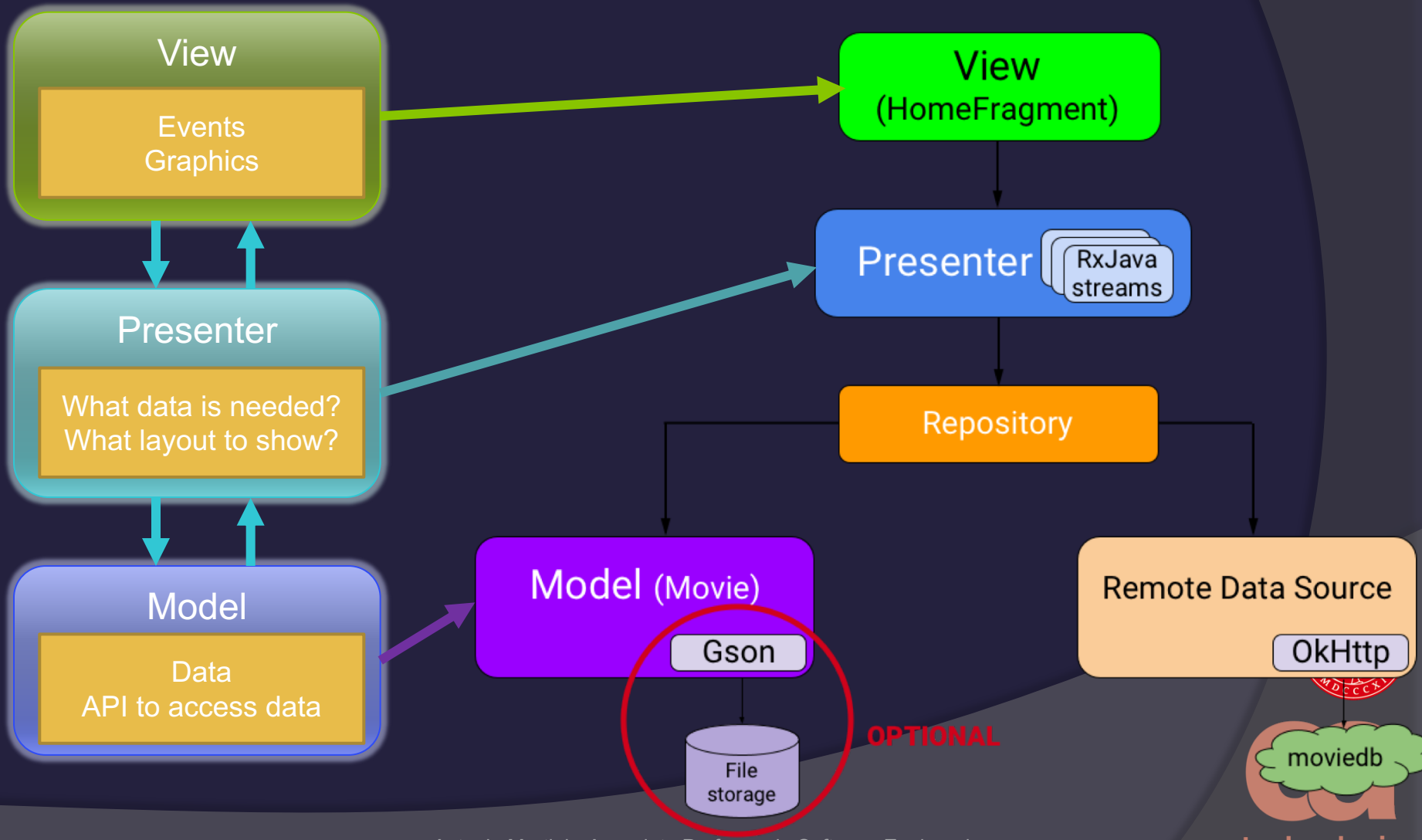
- You might find a lot of guidelines
  - many might be useful
    - but not necessarily for this project
  - some are context-dependent
    - only worth for some kinds of Apps
- Choose wisely!
  - you won't be able to have a perfect architecture the first (few) time(s) you implement an App
  - but it's worth thinking about a few important guidelines from the beginning
- You will see a more hands-on example in the next lecture given by Thomas Lindsjørn

ca technologies

MVP in your App (Android Arch.)



MVP in your App (next lecture)



Technical Debt

Antonio Martini - Associate Professor in Software Engineering

ca technologies

A (classical) conflict among stakeholders

- Sales
  - *“we need to deliver the app fast”*

- Engineers
  - Maintainability

- What should we do?

After the investigation

- Sales
  - *“we need to deliver the app fast”*
  - We need to deliver in 3 months
- Engineers
  - Maintainability
- 2 solutions:
  1. We can deliver in 2 months without a good architecture (MVP)
  2. We can deliver in 3 months and 2 weeks with MVP
- What should we do? How to quantify the cost/benefit?

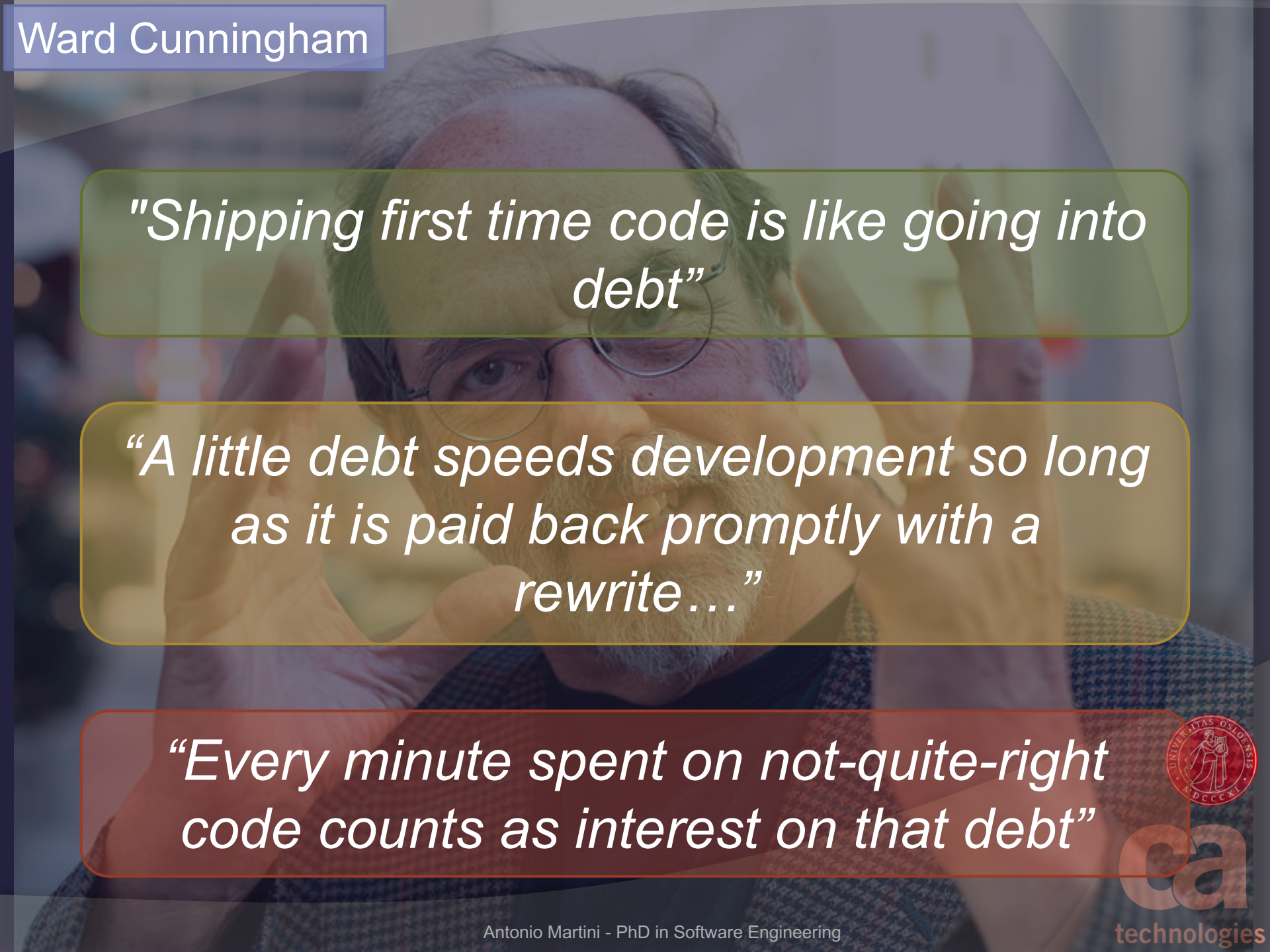
Architectural Technical Debt

- If we take the decision of solution 1, we accumulate Architectural Technical Debt
- But what does it mean?
- Let's start from the beginning…

ca technologies

Ward Cunningham

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"

Current Definition

- *In software-intensive systems, technical debt is a design or implementation construct that is expedient in the short term, but sets up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability*

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

ca technologies

Current Definition

- *In software-intensive systems, technical debt is a design or implementation construct that is expedient in the short term, but sets up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability*

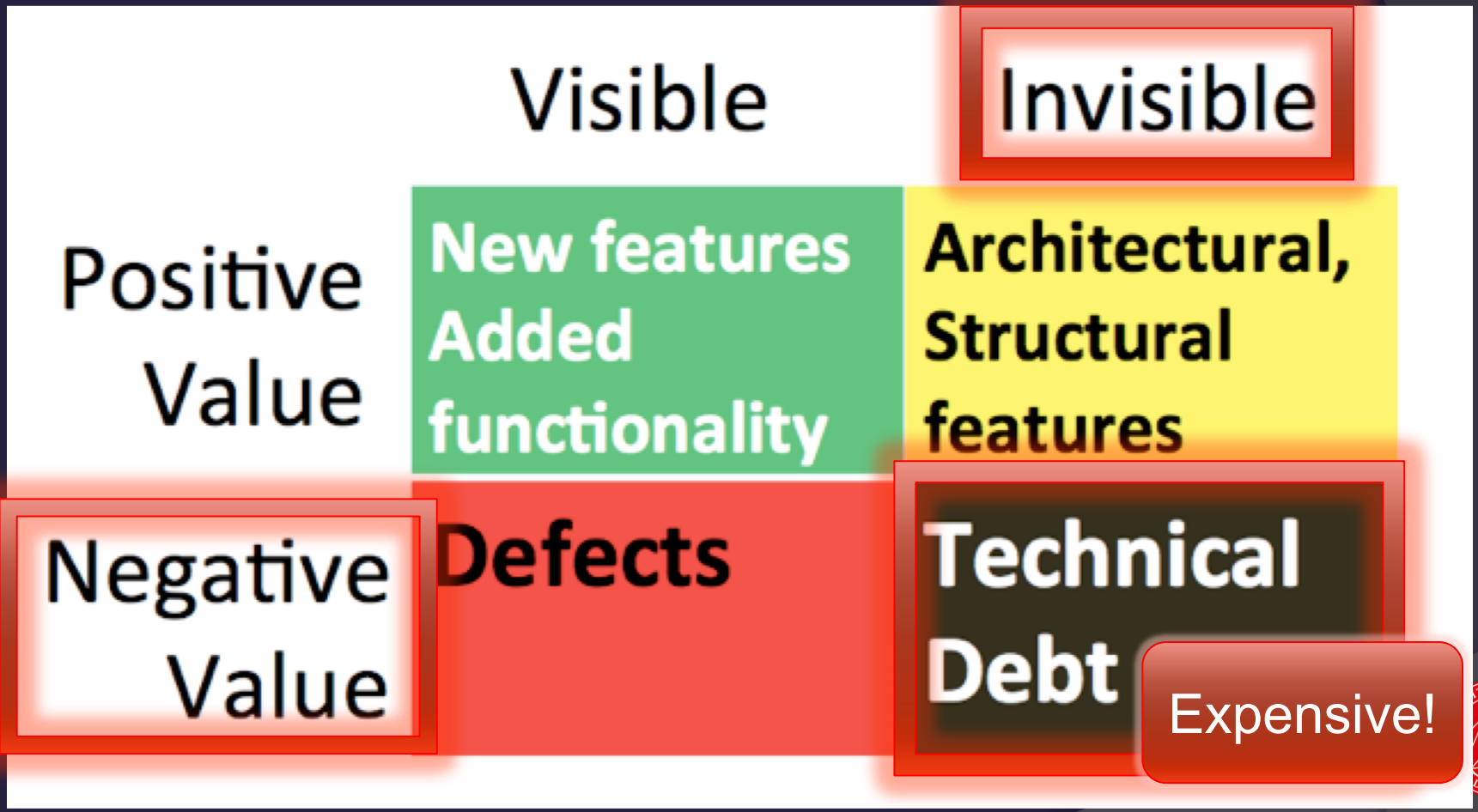
P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, “Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)

ca technologies

What is technical debt?

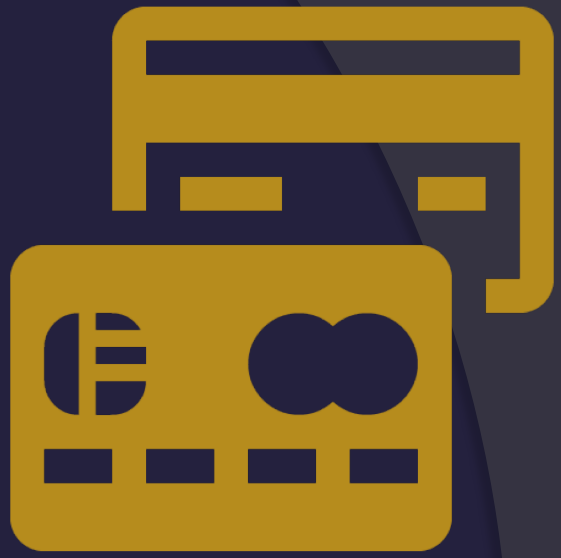
- Debt = sub-optimal solution
  - Save time by non-applying the optimal solution
    - You gain a benefit now (borrow money)
    - but you pay the consequences later (you will pay the interest)

First of all: What is Technical Debt?



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

Credit Card example



- You pay 100 $ at the shop with your credit card instead of using cash
  - You borrow money from the bank
- Next month, you receive the bill of 1100 $
  - The interest is 1000% per month
  - You probably did not know the interest…?
  - Would you have borrow the money with Credit Card if you knew the interest?

Credit Card in Software Development

- Technical sub-optimal solutions are like the debt in the credit card.
- But everyone involved needs to know how much is the interest
- To communicate the risk of high interest when we borrow quality, we use “Technical Debt”

Technical Issue

Think as

Technical Debt

Interest

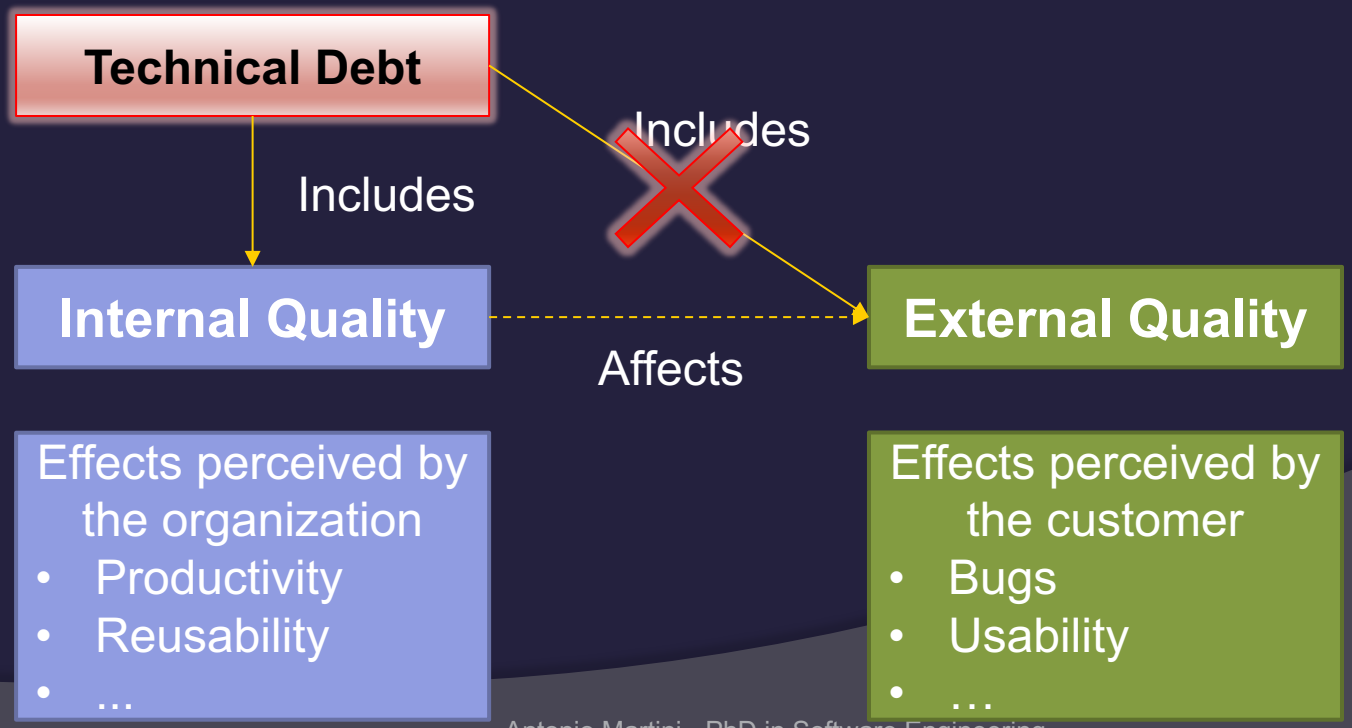
Understand, calculate and communicate

Budget Decisions

Repay the debt

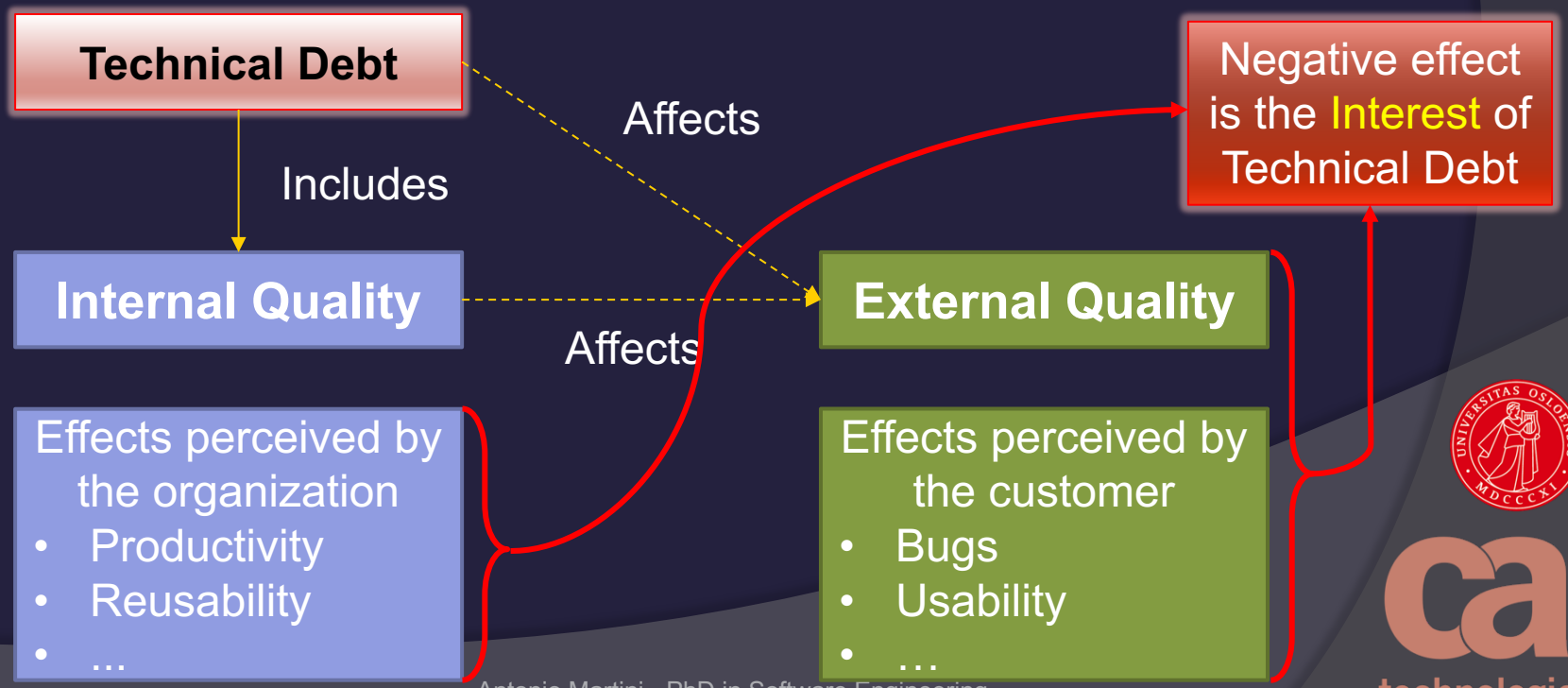
What is technical debt in practice?

- TD includes internal quality issues, not external quality
  - TD is not a bug!
- External quality *might* be influenced by internal quality
  - Example: it might be more difficult to fix a bug *because* of the technical debt

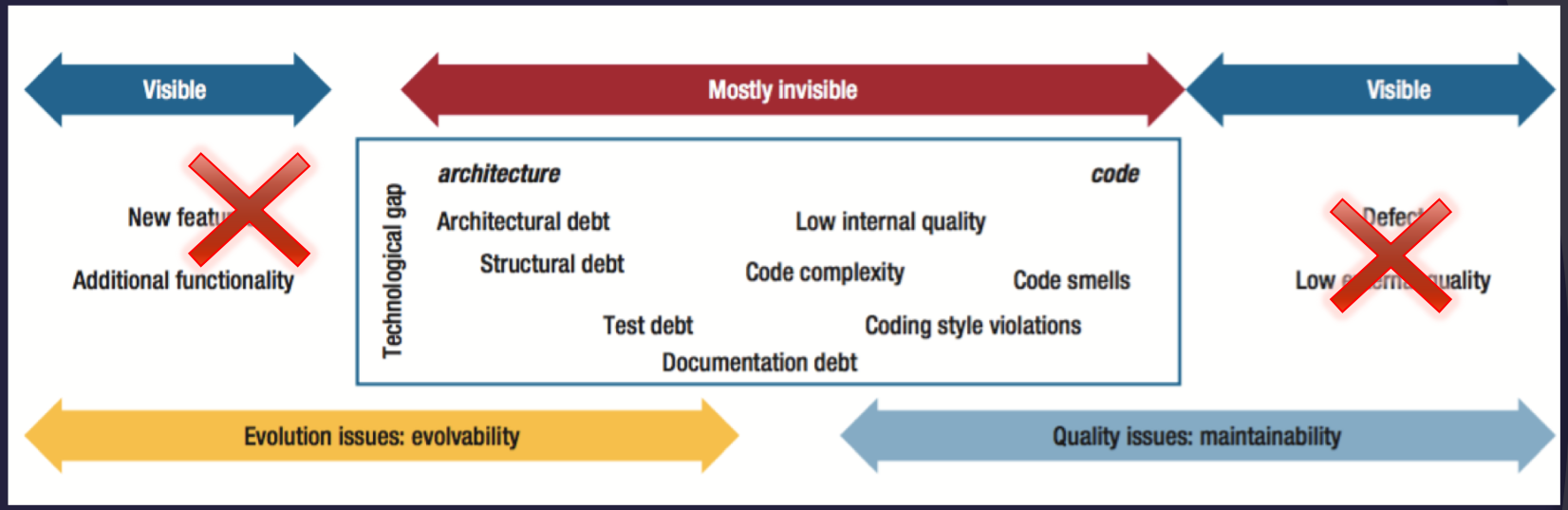


What is technical debt in practice?

- TD includes internal quality issues, not external quality
  - TD is not a bug!
- External quality *might* be influenced by internal quality
  - Example: it might be more difficult to fix a bug *because* of the technical debt



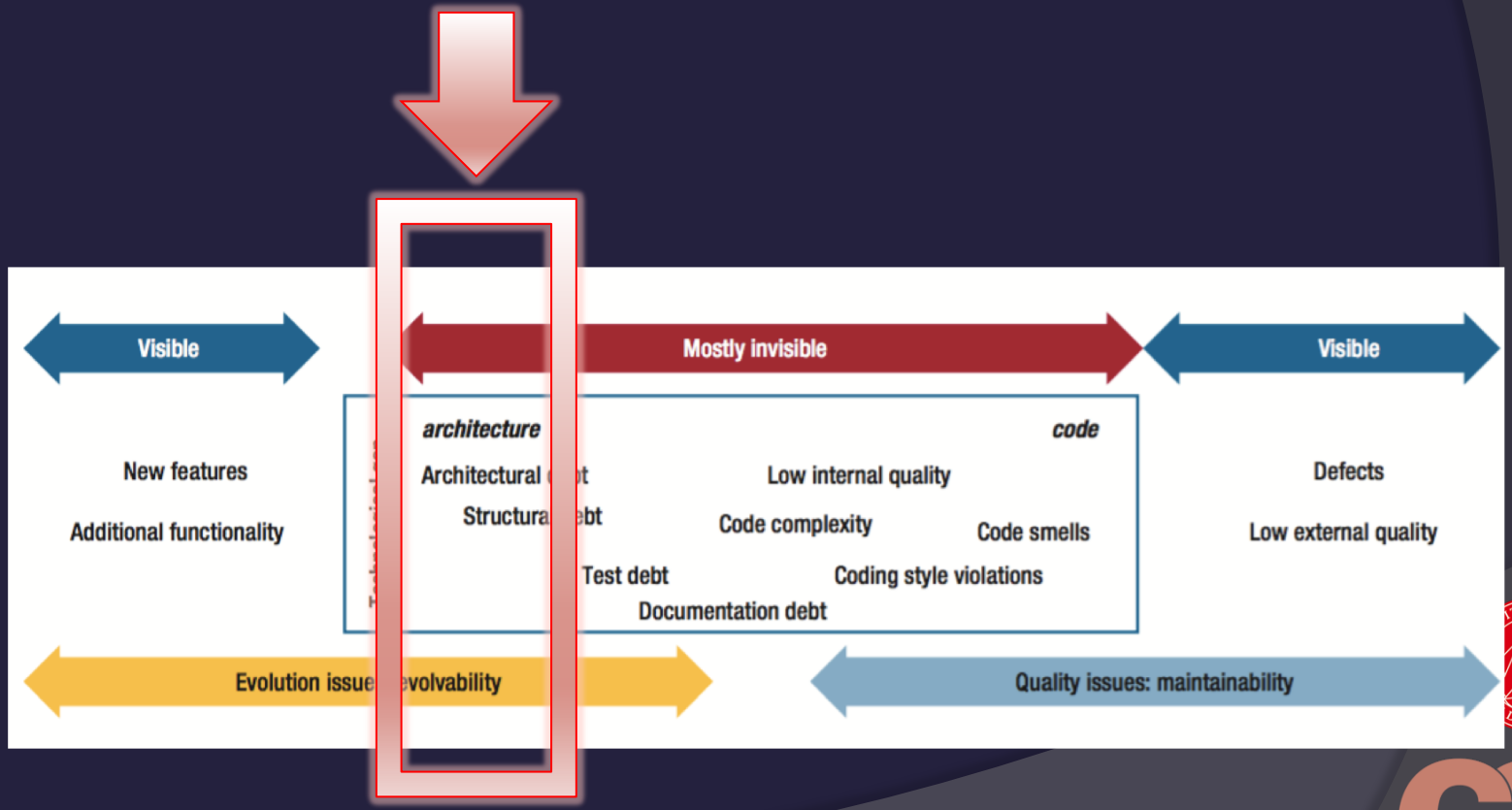
The TD landscape of kinds of TD



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

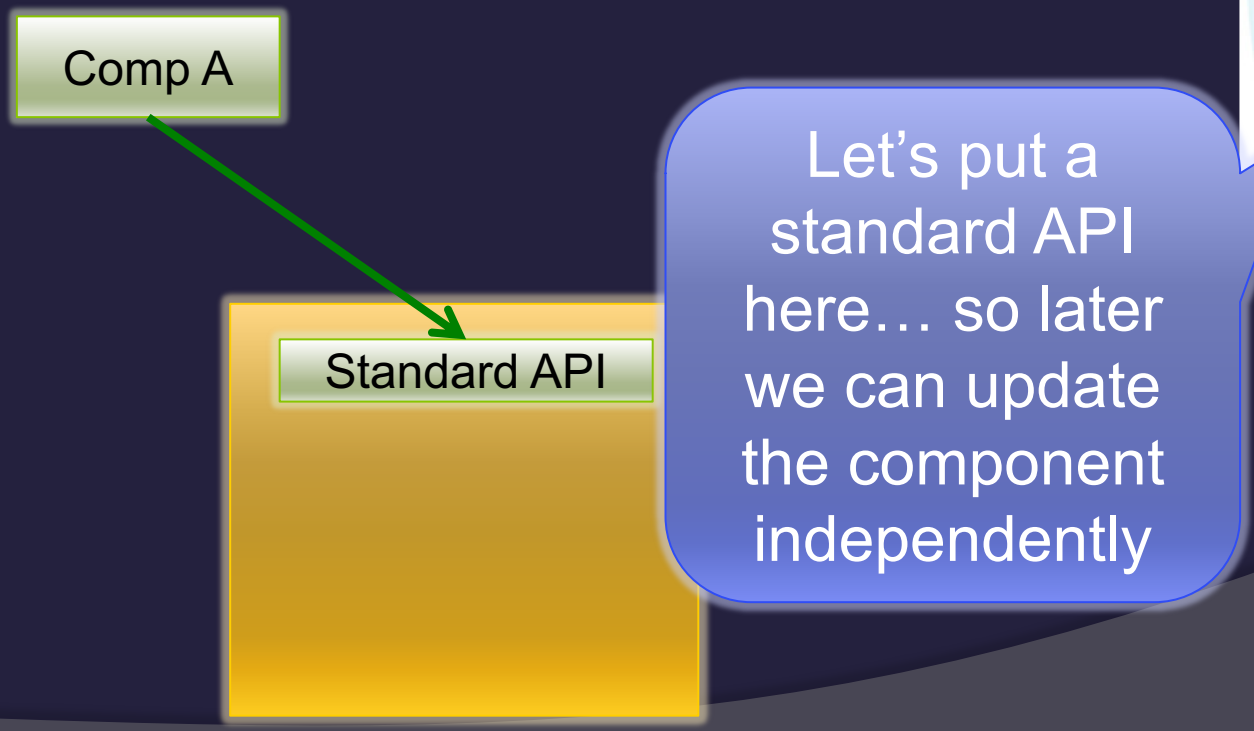
Horror Story

- Technical debt and Architecture



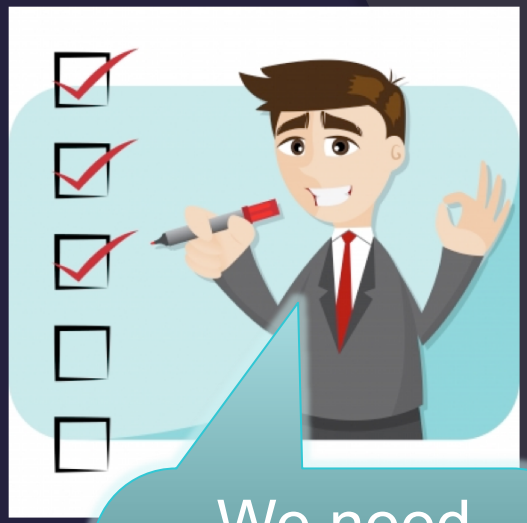
Optimal architectural decision

- Example:
  - Standard public API



During feature development...

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



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

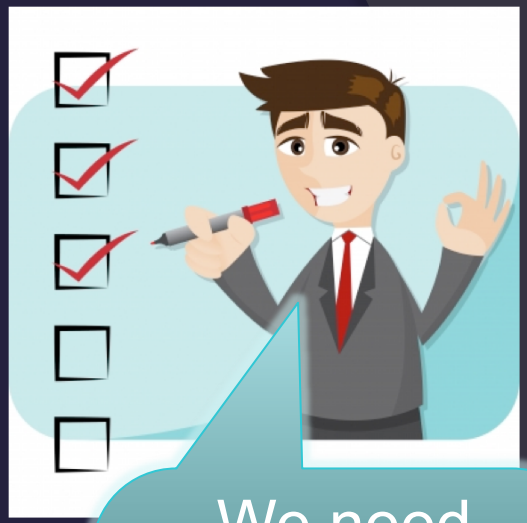
Comp A

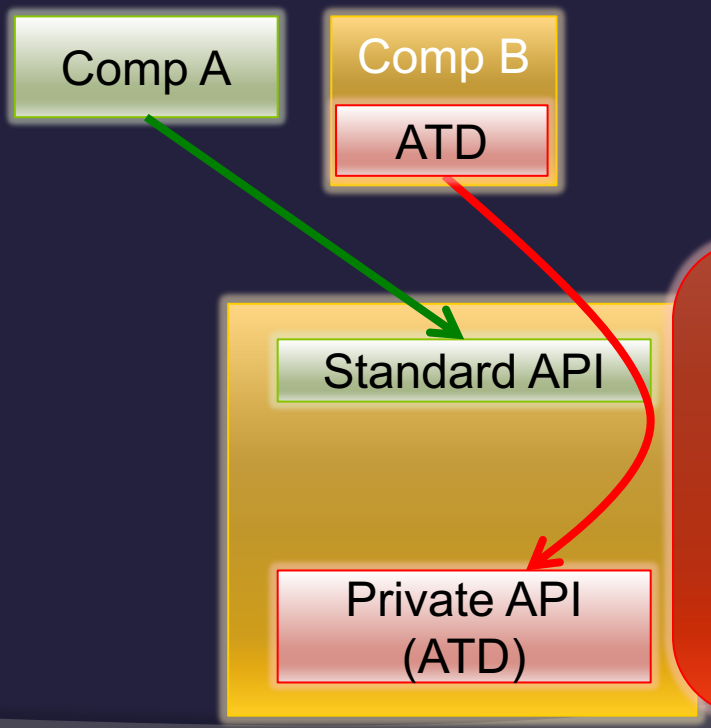
Comp B

Standard API

...with fast delivery comes...

- Deliver fast!





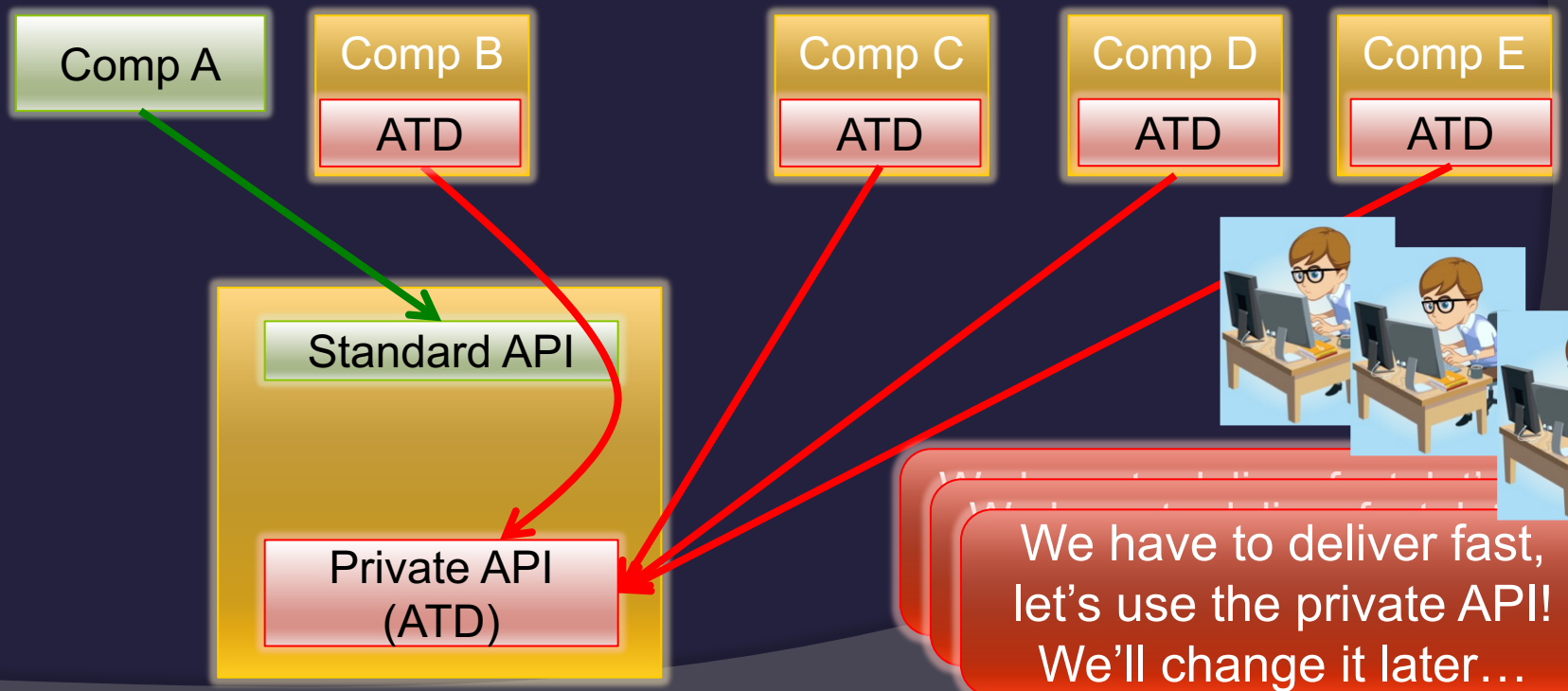
We have to deliver fast, let's use the private API… we'll change it later

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

Fast!

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

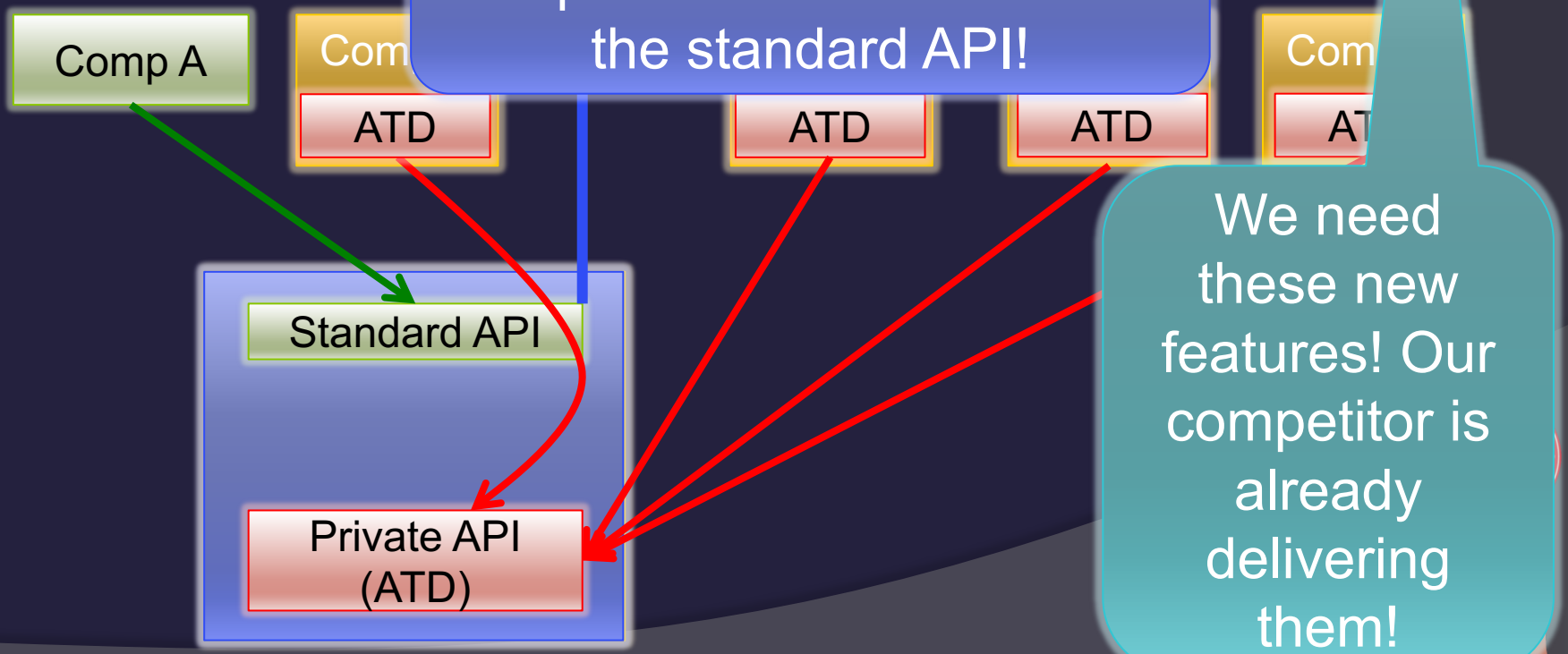
- The violation is spreading to many components



...until, one day...

- New requirement

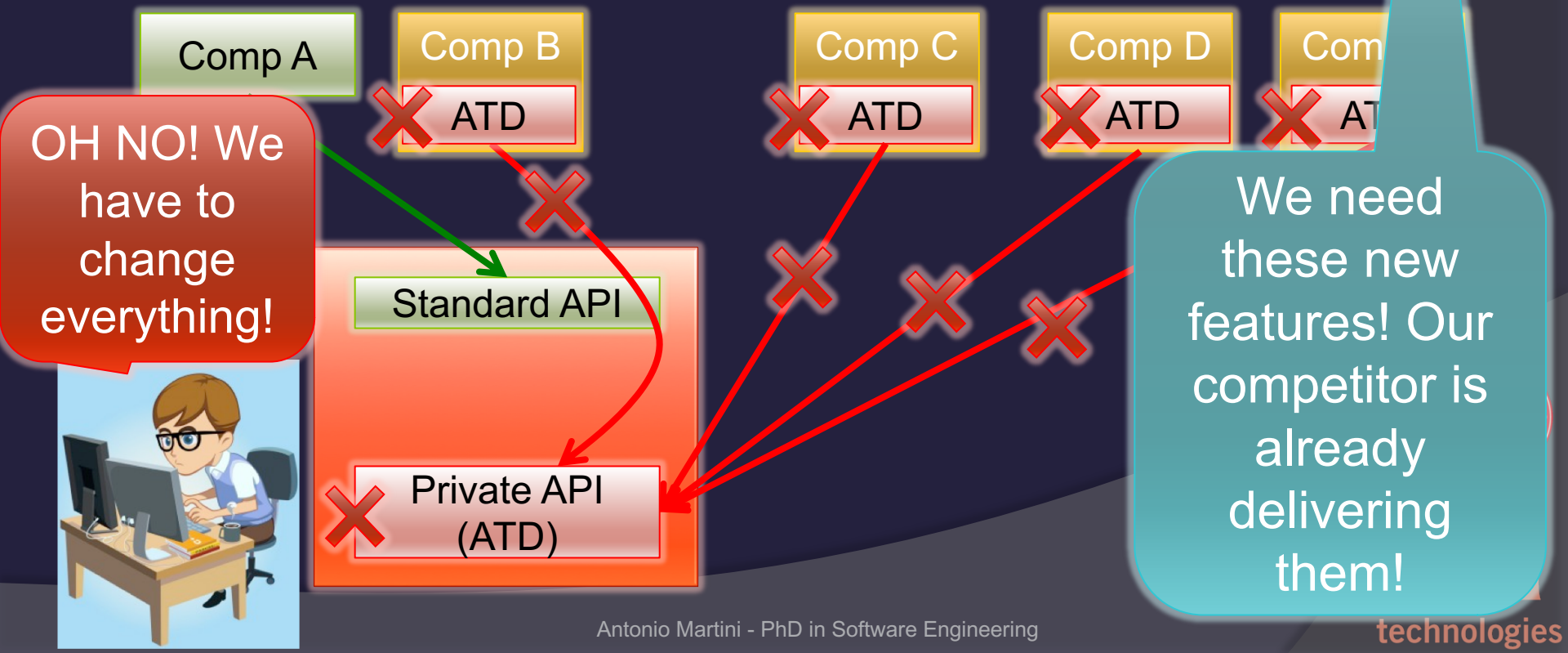
Ok, we can replace this component. The teams used the standard API!



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

...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?

Comp B

ATD

Comp C

ATD

Comp D

ATD

Comp

AT

Standard API

Private API (ATD)

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

So what is Architectural Technical Debt?

- Non-allowed dependencies = "Taking" the Debt
  - Save time by non-applying the optimal solution
- Cost of removing dependencies = Principal
  - How much does it cost to provide the optimal solution?
- Extra evolution cost
  - Replacing the component
- Other impacts = Interest
  - Increasing principal
  - Difficult/Wrong estimation
  - Lead time increases

ca technologies

So what is Architectural Technical Debt?

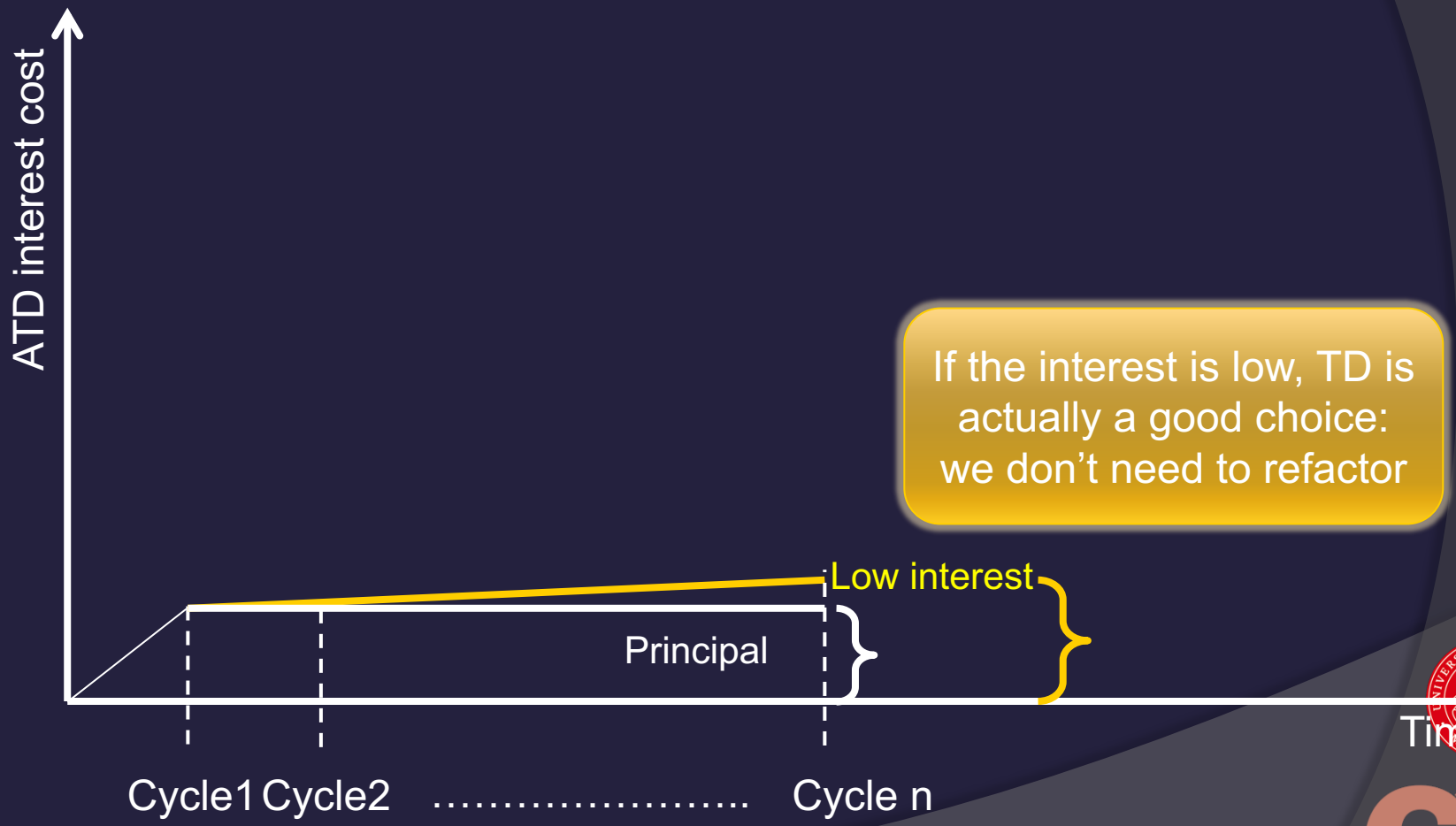
- Non-allowed dependencies = "Taking" the Debt
  - Save time by non-applying the optimal solution
- Cost of removing dependencies = Principal
  - How much does it cost to provide the optimal solution?
- Extra evolution cost
  - Replacing the component
- Other impacts
  - Increasing principal
  - Difficult/Wrong estimation
  - Lead time increases

Important

= Interest

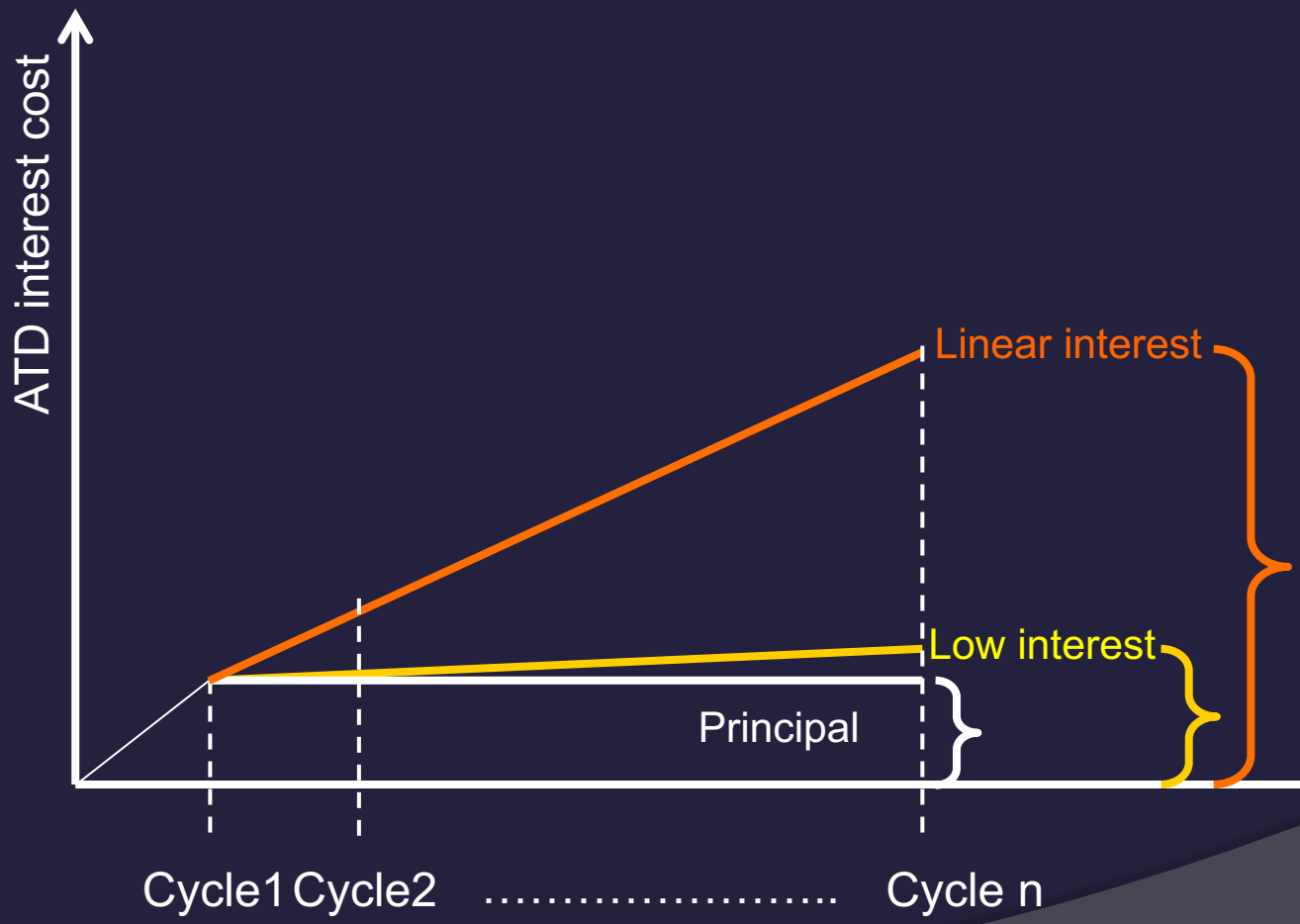
ca technologies

Growing interest

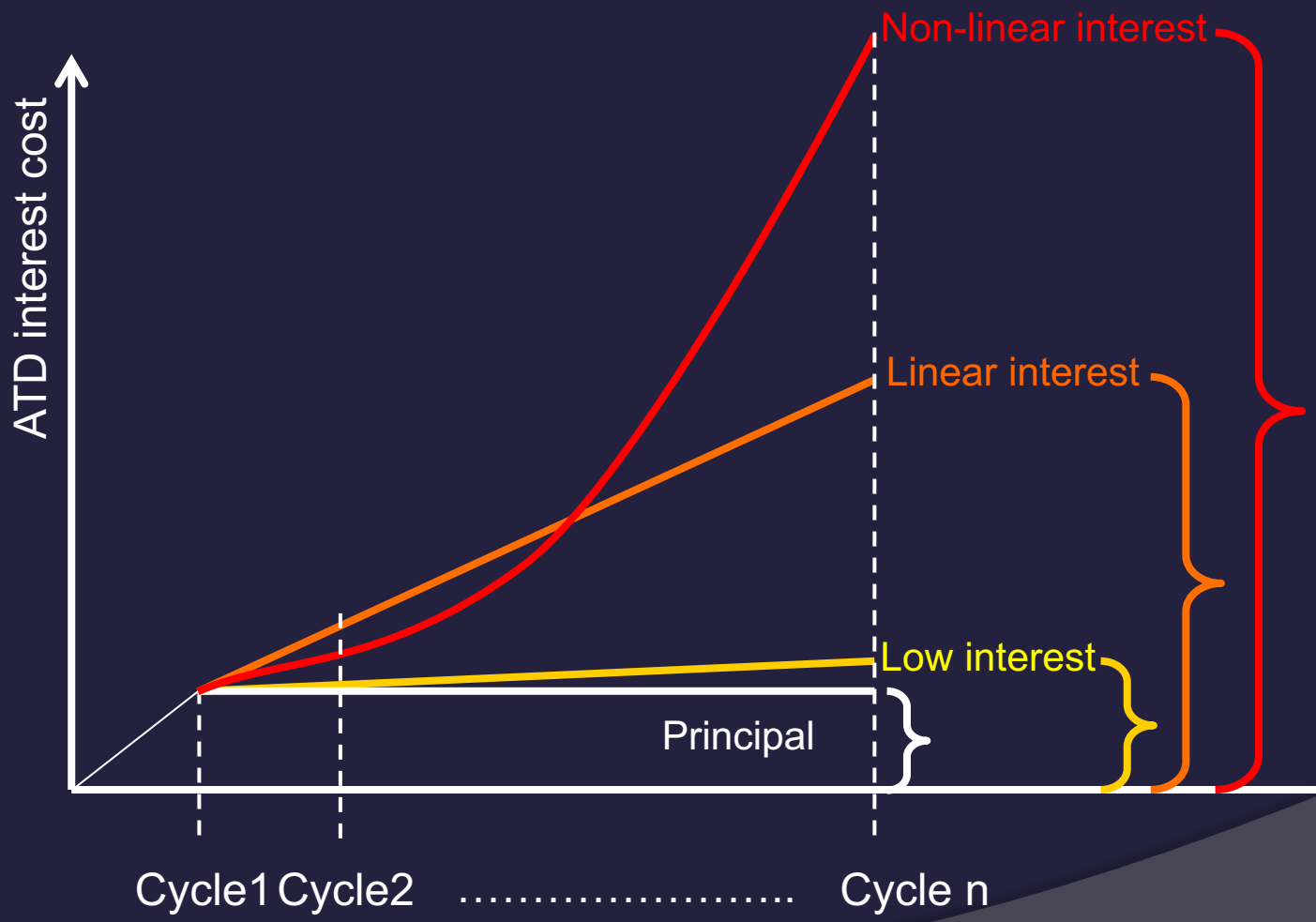


If the interest is low, TD is actually a good choice: we don't need to refactor

Growing interest



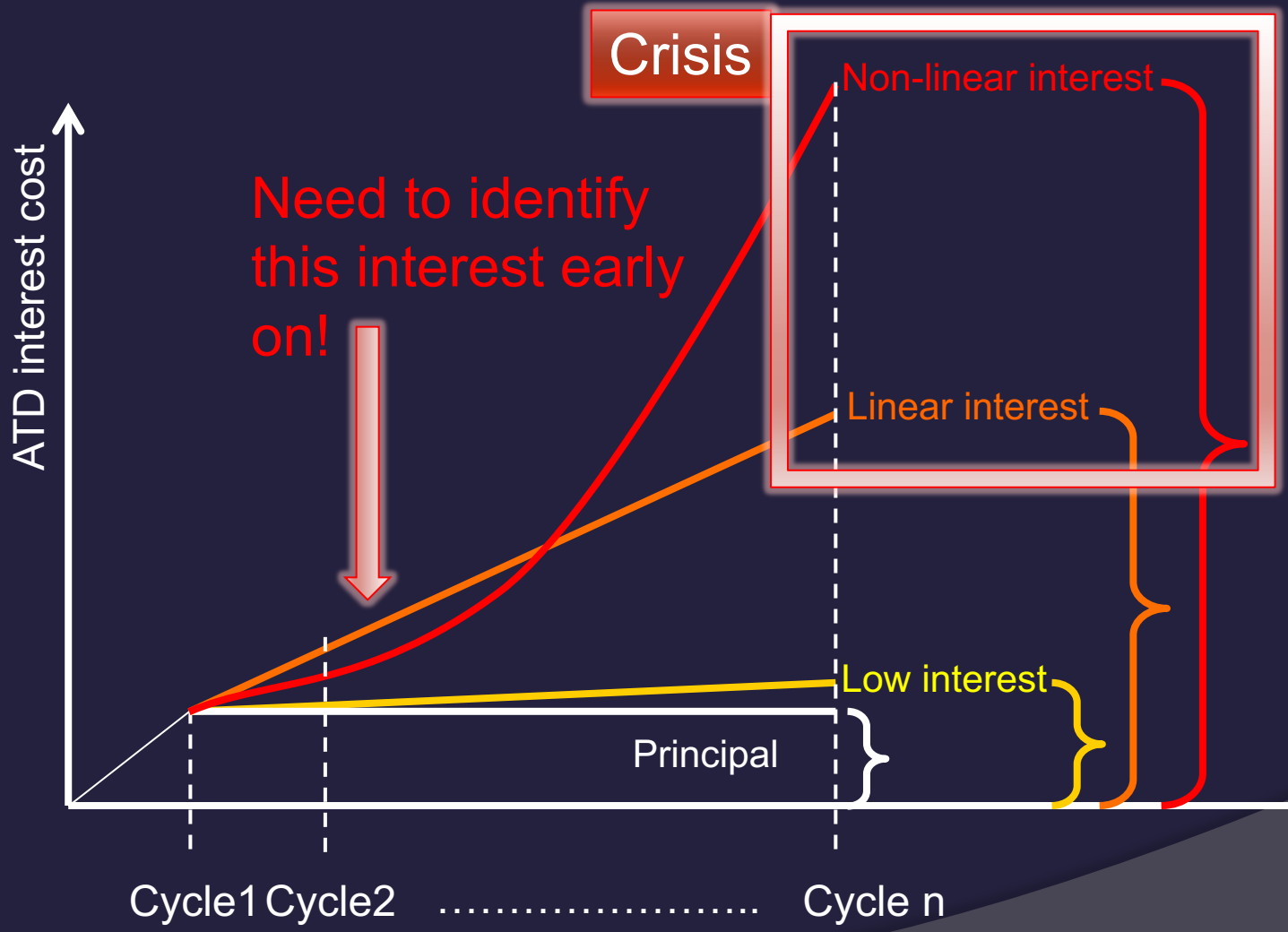
Growing interest



Growing interest

What if it was the interest for the loan?

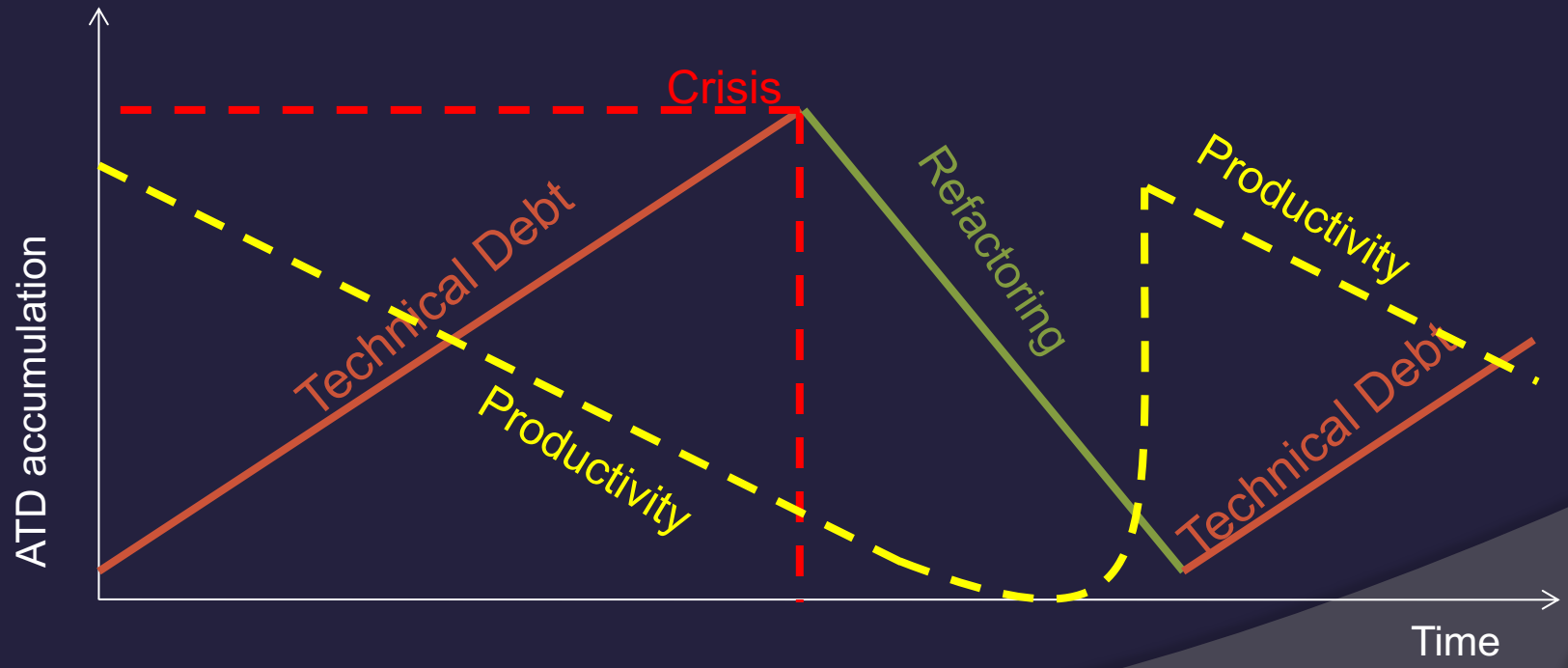
Growing interest



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

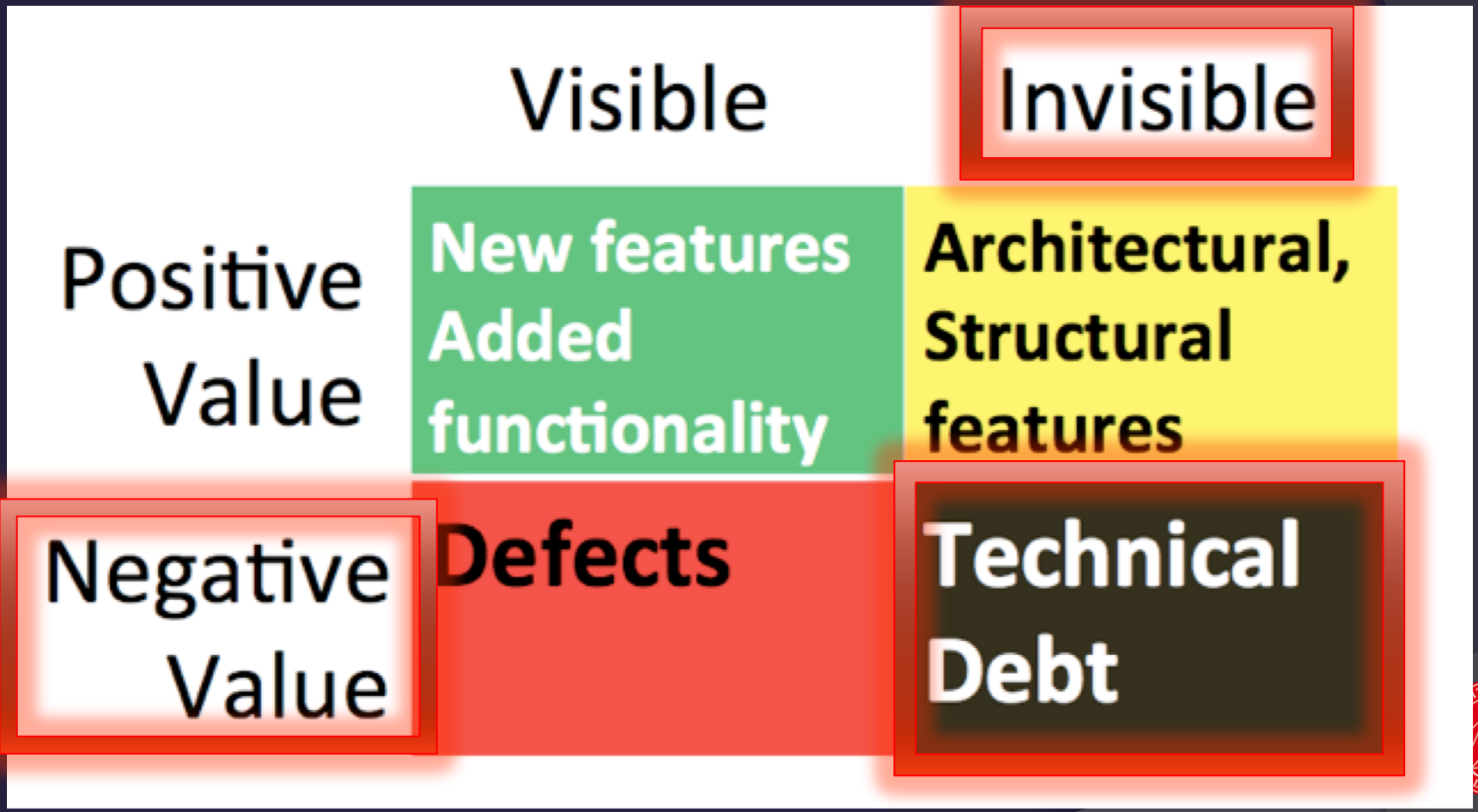
So, what happens in the end?

- Research study in 7 organizations *
- The accumulation of Technical Debt…
- …Leads to crises



* Martini, A., Bosch, J., Chaudron, M., 2015. [1] "Investigating Architectural Technical Debt Accumulation and Refactoring over Time: a Multiple-Case Study," *Information and Software Technology*.

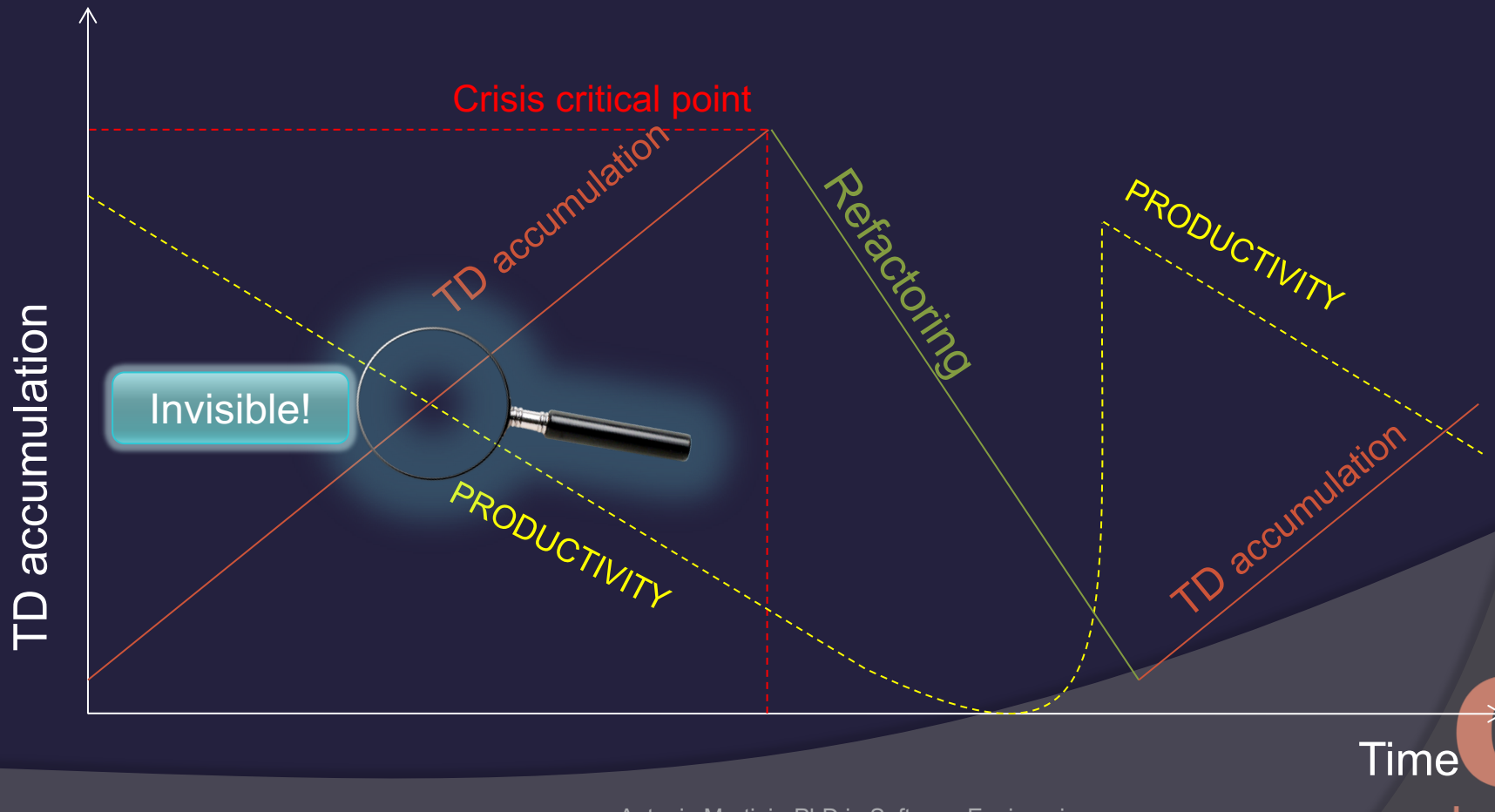
Again, why is TD dangerous?



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

Problem: TD is invisible!

- Invisible accumulation of TD leads to crises



What can we do about TD? Identification

- Once again…don't take debt in the first place!
- Once again…don't implement sub-optimal solutions!

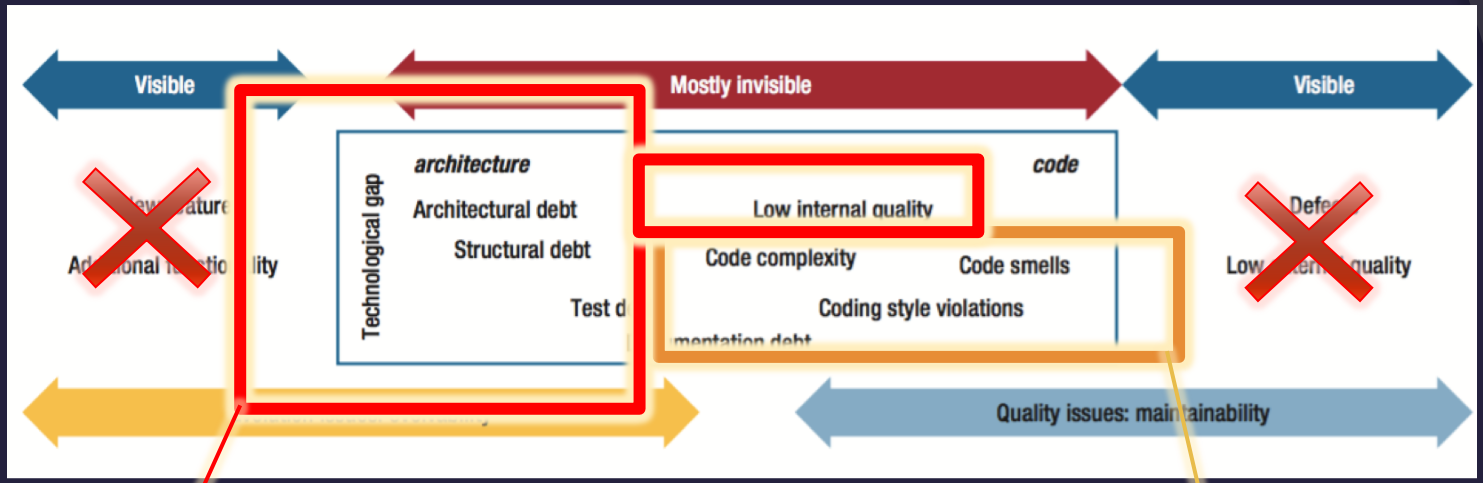
- But in practice you will accumulate some TD. Then, it's important to make it

VISIBLE

- Who will deal with the software that you have developed needs to know the TD

ca technologies

Identification of different kinds of TD



Manual or Invisible

Automatic Tools

(Do not show impact of Technical Debt)

Making TD visible (Identification)

- When you know you are taking debt, create TD items to signal that new TD has been taken
  - Issue tracker
  - Backlog
  - Report the interest of TD! (extra-cost or risk)
- Iteratively check your code to discover TD
- Use available tools
  - SonarQube
    - https://sonarcloud.io/projects?sort=-analysis_date
  - AnaConDebt
    - Or other tracking systems, e.g. Jira
  - Other measures

Let's go back to our conflict...

- Sales
  - *"we need to deliver the app fast"*
  - We need to deliver in 3 months

- Engineers
  - *Maintainability*
  - we want to implement the MVP pattern

- 2 solutions:
  1. We can deliver in 2 months and 2 weeks without a good architecture (MVP)
  2. We can deliver in 3 months and 2 weeks with MVP
- We need to understand the principal and the interest of Technical Debt

Which one to choose?

1. We can deliver in 2 months and 2 weeks without a good architecture (MVP)

- We take Architectural Technical Debt
  - We save 1 month and 2 weeks now
  - We will have to refactor later (principal)
    - Let's say other 3 months (rewrite from scratch)
  - The interest is high every time we add a new feature:
    - High testability costs
    - High maintainability costs
    - Can we quantify them?
      - E.g. in six months we will add 6 features, and we will spend, for each, 1 additional week
      - We have to add 1.5 months of waste

2. We can deliver in 3 months and 2 weeks with MVP
   - Is it a problem to deliver 2 weeks later?

ca technologies

Scenarios and analysis

	Benefit: Users short-term	Cost: Interest long-term	Cost	Total
Solution 1	+ (vs competitor)	- (high interest)	- (3 months refactoring needed)	-1
Solution 2	- (vs competitor)	+ (saved interest)	+ (we don't have to refactor)	+1

We choose to not accumulate Technical Debt

- It's more convenient!
- But not always… sometimes, Technical Debt can be useful

ca technologies

Technical Debt in your project:

- Decide what Technical Debt to take or not
- If took TD during the project, document it by logging:
  - Technical Debt Items
    - Mention the estimates for
      - Cost of Refactoring (Principal)
      - Extra-costs (Interest)
- Deliver the document together with the project

Take aways

ca technologies

Don’t forget about Architecture!

Communicate with the Stakeholders

Follow Business goals, not dogmas

Take Technical Debt only if necessary

If you must take Technical Debt, make it visible

Questions?

Comments?

- antonio.martini@ifi.uio.no

technologies