Antonio Martini

Professor in Software Engineering

University of Oslo

Course IN2000
2024-02-13

ARCHITECTURE AND TECHNICAL DEBT

Who is Antonio Martini?

- Italian
  - No kebab pizza! ☺
  - 6 years in Sweden, 6 in Norway – survived many winters!

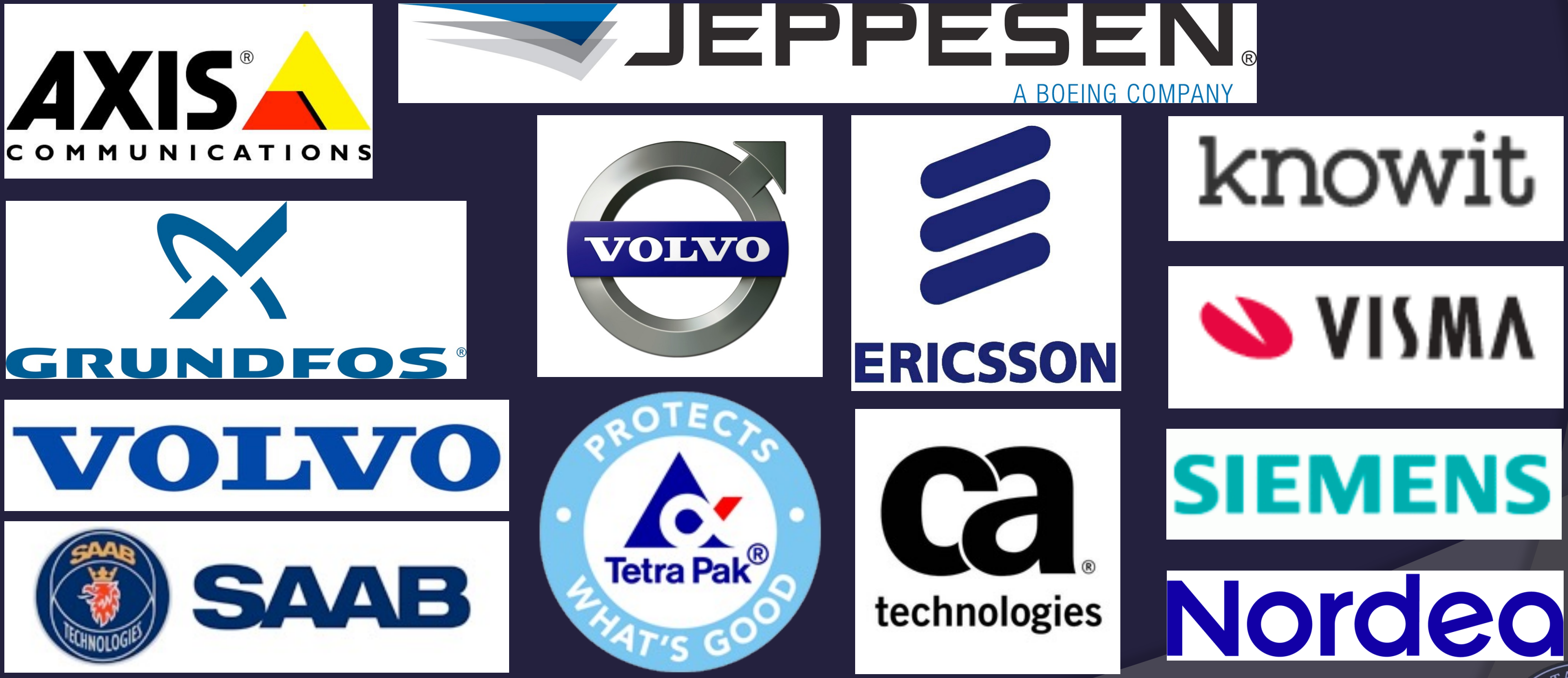
- Worked as a Software Developer
- PhD in Software Engineering at Chalmers
- Principal Strategic Researcher at CA Technologies
- Independent consultant
  - Ericsson, Volvo IT, etc.
  - AnaConDebt tool

- Currently:
  - Associate Professor at University of Oslo
  - Startup founder ACDtek

- Hobbies
  - Board games, strategy computer games, pool, etc.
  - Football, volleyball, beach volley, fencing
  - Piano, Drumset, etc.
  - Travel!
  - …and no time for them! ☺

Several projects on architecture and technical debt

Some collaborators from industry:



Agenda

- What is software architecture?
- Thinking about architecture
  - Stakeholder analysis
  - Trade-offs
- Principles of Software Architecture
  - Components and APIs
  - Design tradeoffs
  - Architectural styles
- Intro to Technical Debt

- Summary

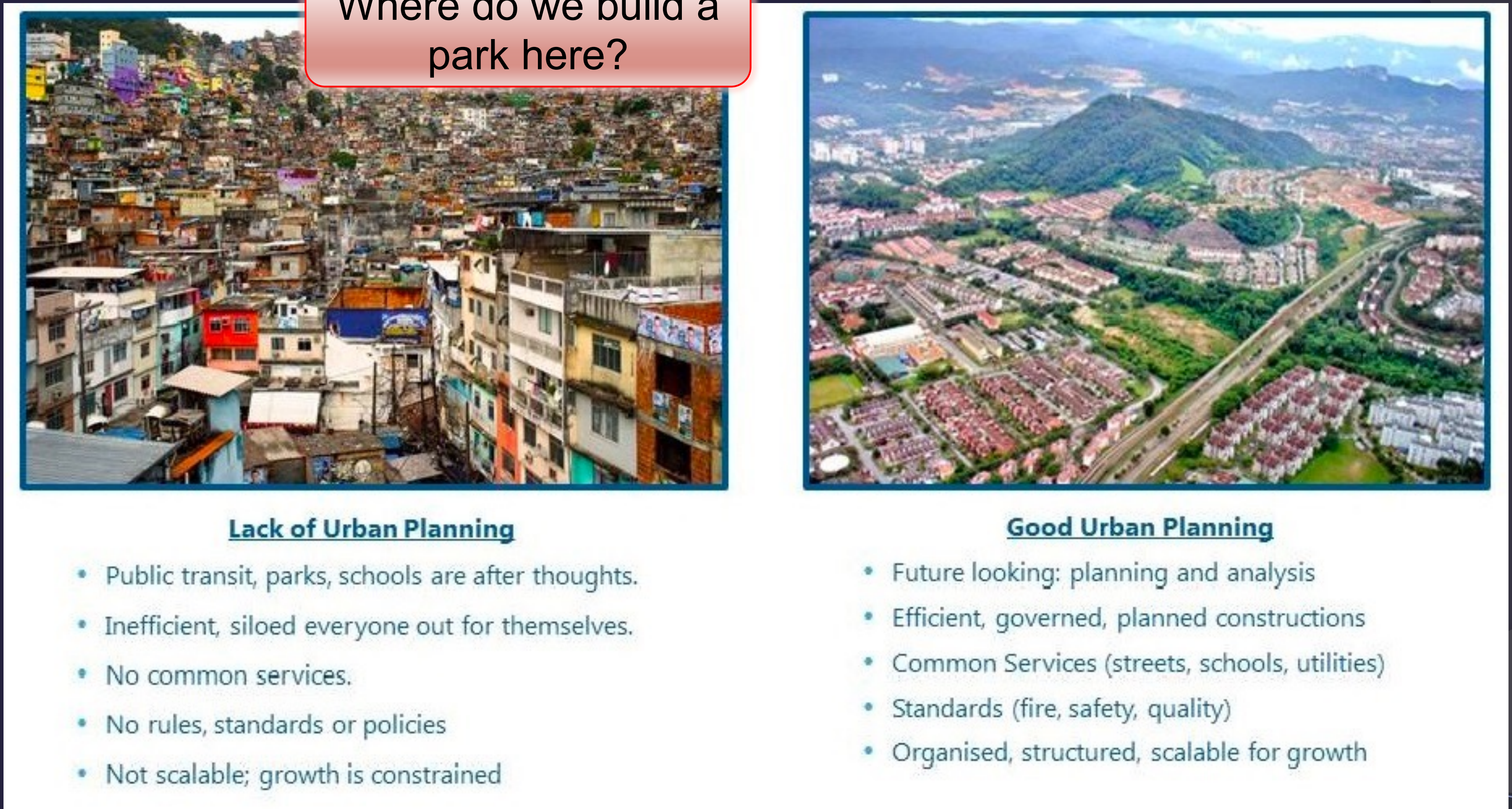
- Interacting questions during the lecture ?

- Relevant for the project and activities PROJECT

What is Software Architecture?

What’s the difference?

Where do we build a park here?



Lack of Urban Planning

- Public transit, parks, schools are after thoughts.
- Inefficient, siloed everyone out for themselves.
- No common services.
- No rules, standards or policies
- Not scalable; growth is constrained

Good Urban Planning

- Future looking: planning and analysis
- Efficient, governed, planned constructions
- Common Services (streets, schools, utilities)
- Standards (fire, safety, quality)
- Organised, structured, scalable for growth

Software architecture is...

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

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

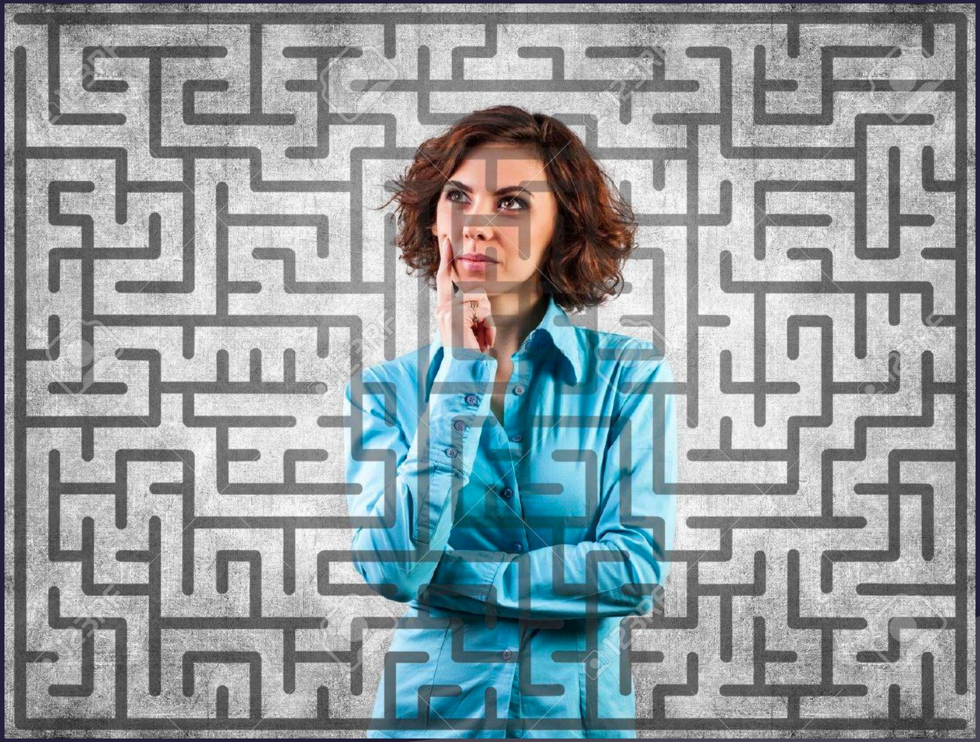
You can't ignore architecture

- All products HAVE an architecture
  - It can be bad
  - It can be good
- In all projects we SHOULD think about architecture
  - Maybe less in small projects
  - Maybe more in large projects
- Thinking about the architecture is a necessary (and smart) process

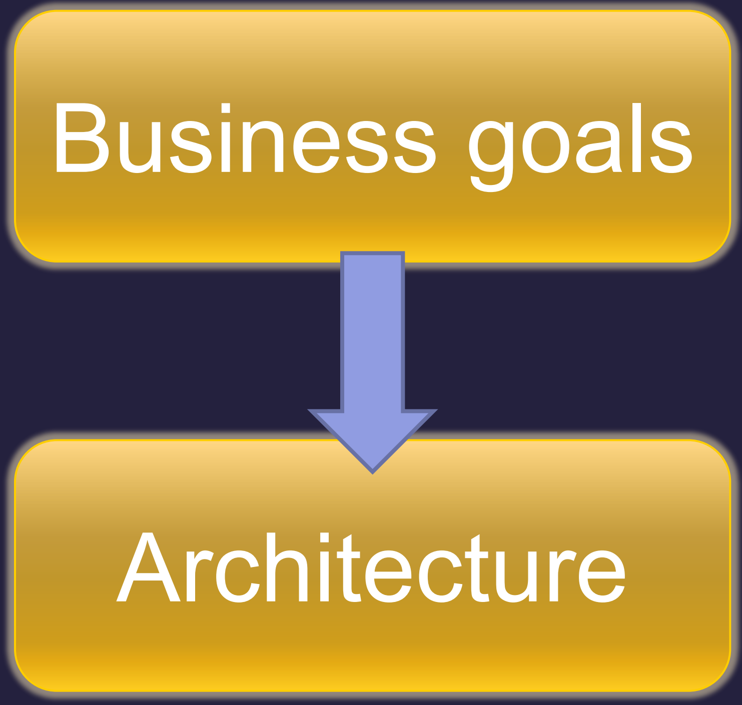
How to think about Architecture

How to choose an architecture

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



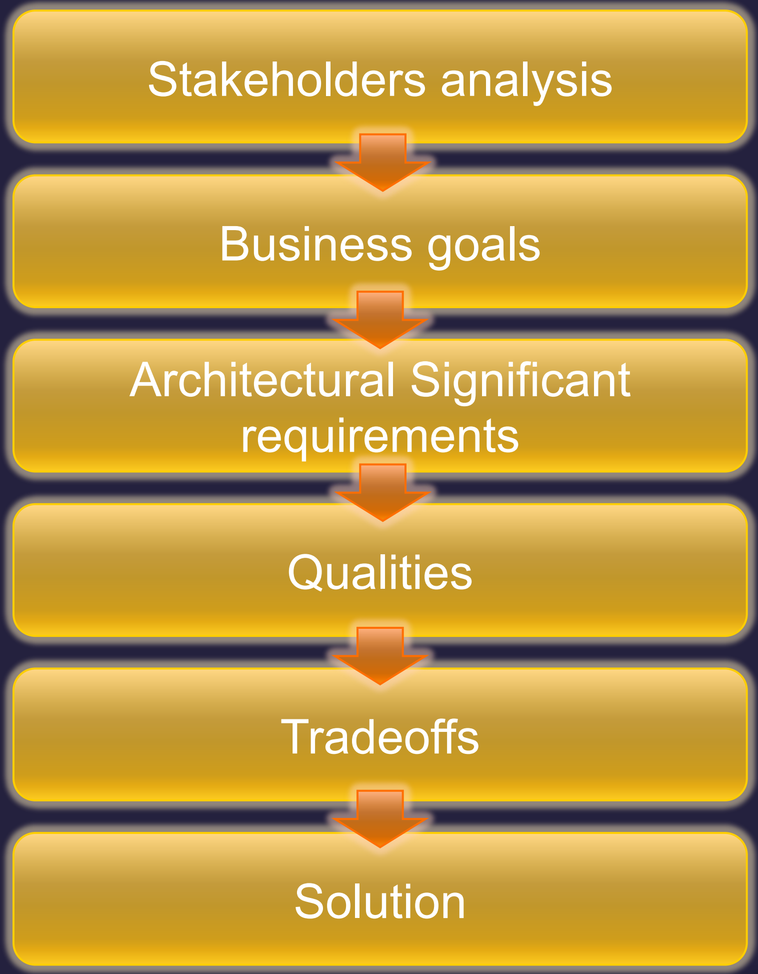
Business drives architecture



Business goals

Architecture

A process to think about architecture



Stakeholders analysis

Business goals

Architectural Significant requirements

Qualities

Tradeoffs

Solution

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 for a game app like Pokemon Go?
  - What are their needs?
  - Write down 2
    - <Stakeholder> : <Need>

Stakeholder analysis (2)

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

Needs examples

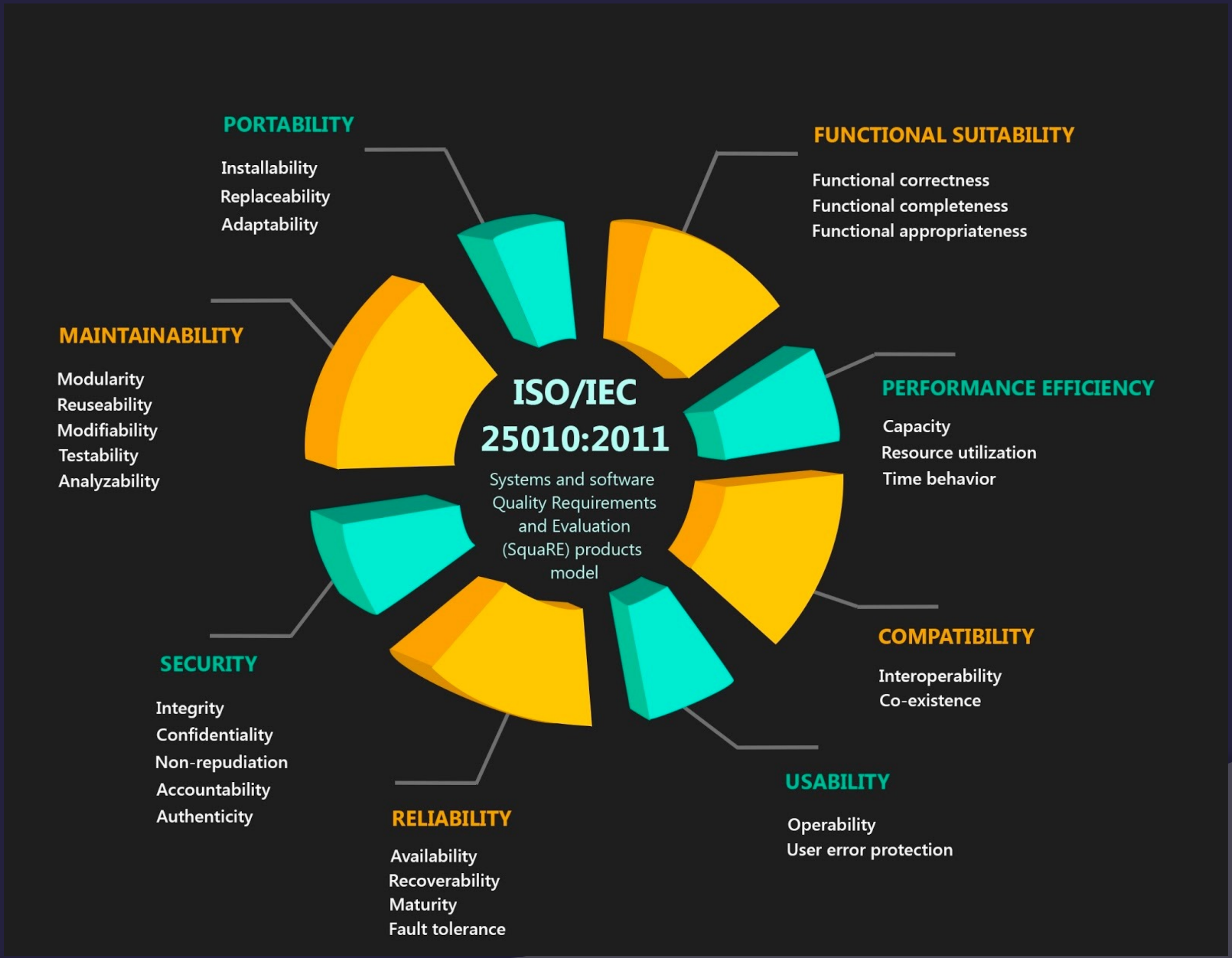
- Sales' needs:
  - *"we need to deliver the app fast"*
  - *"we need the app to be available for both Android and iOS"*

- Users' needs
  - *"we want to have an experience without bugs"*
  - *"we want it to get the information in real time"*

- Engineers' needs
  - *"we need to test the app easily"*
  - *"we need to be able to deploy new features quickly after the first release"*

APPROVED

System Qualities



ISO/IEC 25010:2011

Systems and software Quality Requirements and Evaluation (SquaRE) products model

FUNCTIONAL SUITABILITY

- Functional correctness
- Functional completeness
- Functional appropriateness

PERFORMANCE EFFICIENCY

- Capacity
- Resource utilization
- Time behavior

COMPATIBILITY

- Interoperability
- Co-existence

USABILITY

- Operability
- User error protection

RELIABILITY

- Availability
- Recoverability
- Maturity
- Fault tolerance

SECURITY

- Integrity
- Confidentiality
- Non-repudiation
- Accountability
- Authenticity

MAINTAINABILITY

- Modularity
- Reuseability
- Modifiability
- Testability
- Analyzability

PORTABILITY

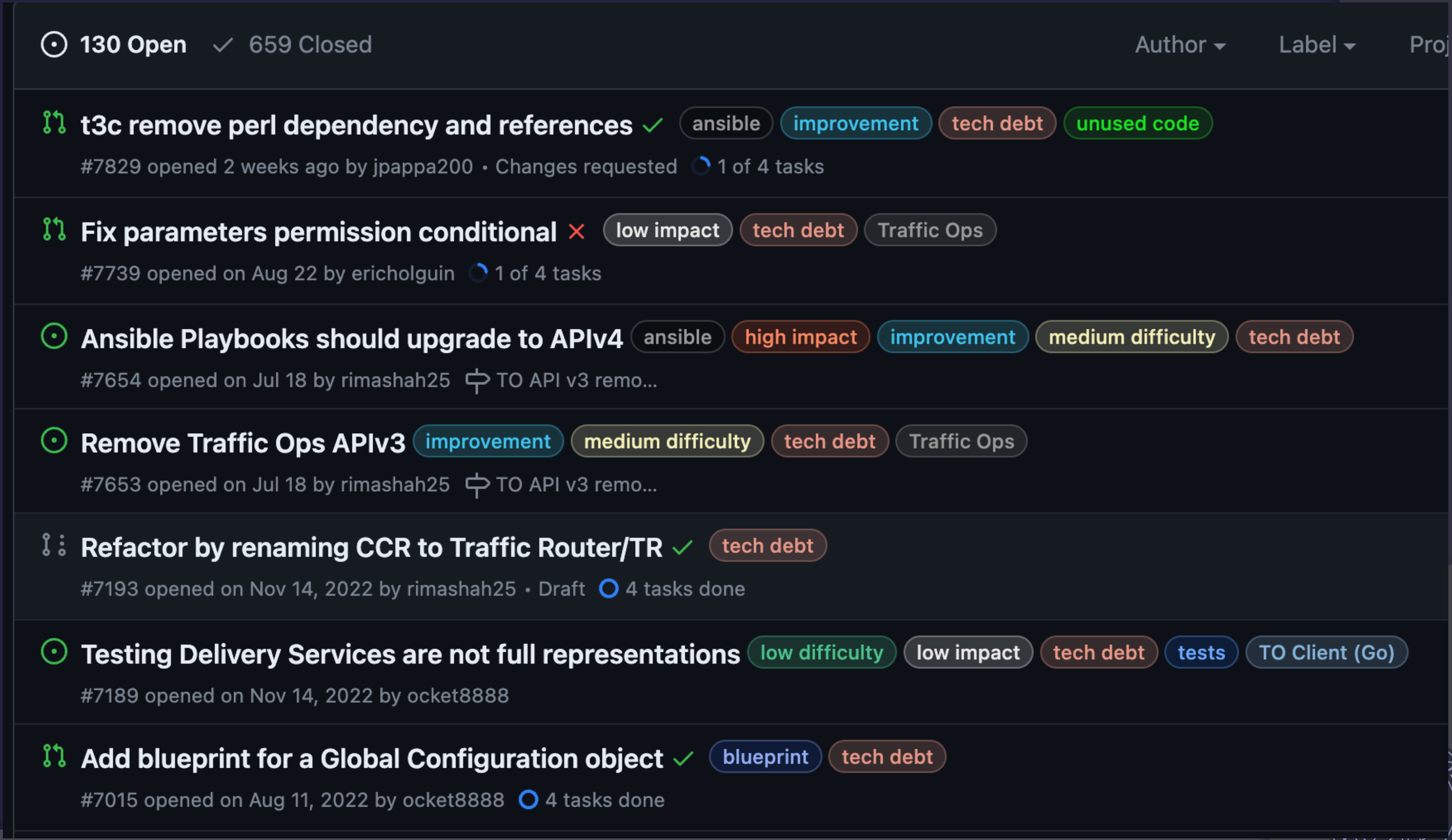
- Installability
- Replaceability
- Adaptability

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

Qualities (non-functional)

- Maintainability - the ease with which a product can be maintained
  - E.g. Fix defects, meet new requirements, etc.
- Performance – how efficiently software can perform a task
  - E.g. How long does it take to load a web-page?
- Security – how solid the system is in protecting from attacks by malicious actors or by disruptions
  - E.g. Confidential data leaks
- Reliability - ability of equipment to function without failure
  - E.g. Bugs
- Usability - perform the tasks safely, effectively, and efficiently while enjoying the experience
  - E.g. Easy-to-use UI
- Compatibility - the ability of software and hardware from different sources to work together without having to be altered to do so
  - E.g. New software that runs on older cars
- Portability - easily made to run on different platfotms
  - E.g. Android, IOS, etc.

Tag your tasks



○ 130 Open ✓ 659 Closed

Author ▾ Label ▾ Proj

t3c remove perl dependency and references ✓ ansible improvement tech debt unused code

#7829 opened 2 weeks ago by jpappa200 • Changes requested ◔ 1 of 4 tasks

Fix parameters permission conditional ✗ low impact tech debt Traffic Ops

#7739 opened on Aug 22 by ericholguin ◔ 1 of 4 tasks

Ansible Playbooks should upgrade to APIv4 ansible high impact improvement medium difficulty tech debt

#7654 opened on Jul 18 by rimashah25 TO API v3 remo...

Remove Traffic Ops APIv3 improvement medium difficulty tech debt Traffic Ops

#7653 opened on Jul 18 by rimashah25 TO API v3 remo...

Refactor by renaming CCR to Traffic Router/TR ✓ tech debt

#7193 opened on Nov 14, 2022 by rimashah25 • Draft ○ 4 tasks done

Testing Delivery Services are not full representations low difficulty low impact tech debt tests TO Client (Go)

#7189 opened on Nov 14, 2022 by ocket8888

Add blueprint for a Global Configuration object ✓ blueprint tech debt

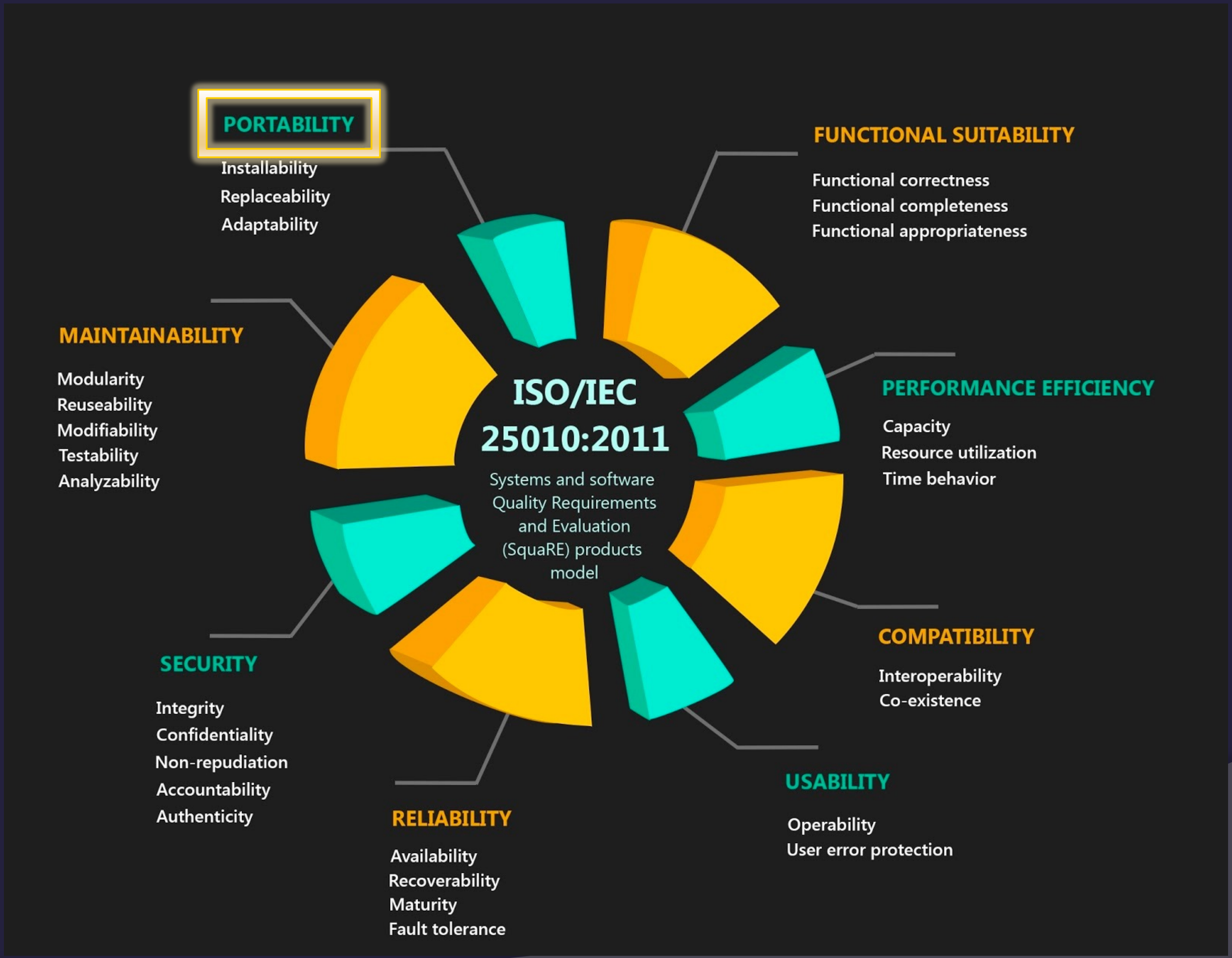
#7015 opened on Aug 11, 2022 by ocket8888 ○ 4 tasks done

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 – Time constraint*
  2. *Portability*

System Qualities - Sales



ISO/IEC 25010:2011

Systems and software Quality Requirements and Evaluation (SquaRE) products model

PORTABILITY
- Installability
- Replaceability
- Adaptability

FUNCTIONAL SUITABILITY
- Functional correctness
- Functional completeness
- Functional appropriateness

MAINTAINABILITY
- Modularity
- Reuseability
- Modifiability
- Testability
- Analyzability

PERFORMANCE EFFICIENCY
- Capacity
- Resource utilization
- Time behavior

COMPATIBILITY
- Interoperability
- Co-existence

SECURITY
- Integrity
- Confidentiality
- Non-repudiation
- Accountability
- Authenticity

USABILITY
- Operability
- User error protection

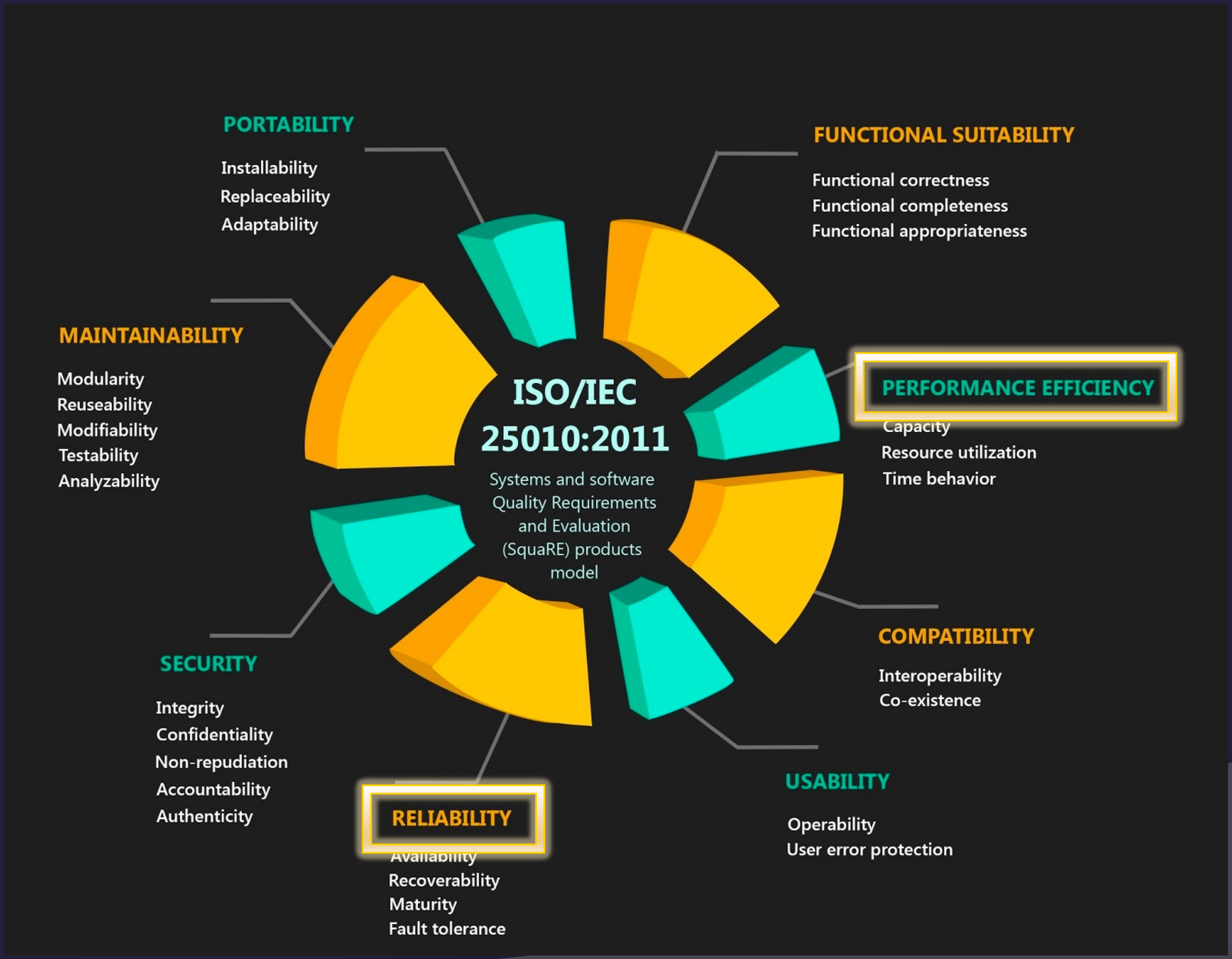
RELIABILITY
- Availability
- Recoverability
- Maturity
- Fault tolerance

From needs to qualities - users

- Users' needs
  1. *"we want to have an experience without bugs"*
  2. *"we want it to see the real time results quickly"*

- Qualities?
  1. *Reliability*
  2. *Performance*

System Qualities – Users



ISO/IEC 25010:2011

Systems and software Quality Requirements and Evaluation (SquaRE) products model

PORTABILITY

- Installability
- Replaceability
- Adaptability

FUNCTIONAL SUITABILITY

- Functional correctness
- Functional completeness
- Functional appropriateness

MAINTAINABILITY

- Modularity
- Reuseability
- Modifiability
- Testability
- Analyzability

PERFORMANCE EFFICIENCY

- Capacity
- Resource utilization
- Time behavior

COMPATIBILITY

- Interoperability
- Co-existence

SECURITY

- Integrity
- Confidentiality
- Non-repudiation
- Accountability
- Authenticity

RELIABILITY

- Availability
- Recoverability
- Maturity
- Fault tolerance

USABILITY

- Operability
- User error protection

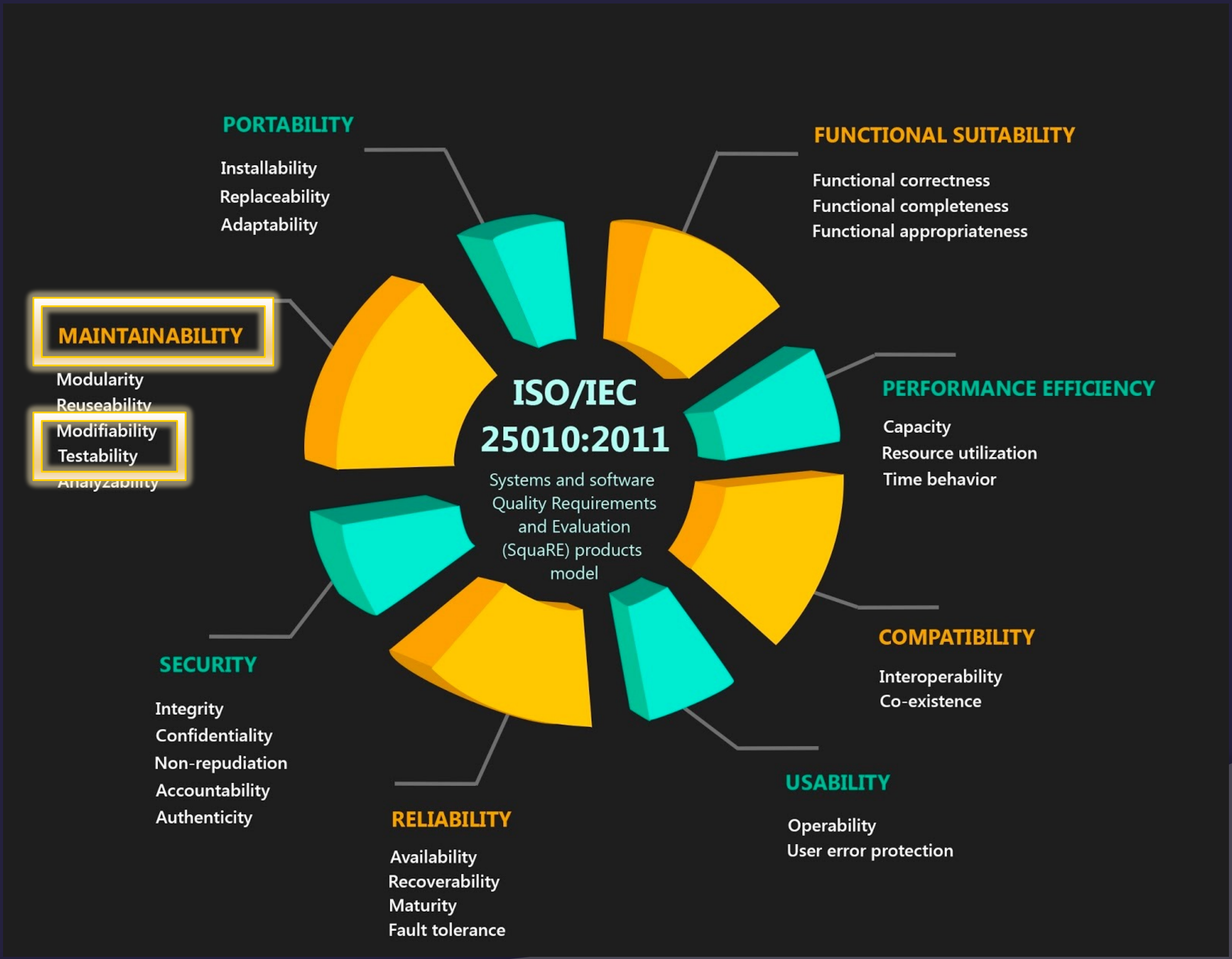
From needs to qualities - engineers

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

APPROVED

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

System Qualities - Engineers



ISO/IEC 25010:2011

Systems and software Quality Requirements and Evaluation (SquaRE) products model

PORTABILITY

- Installability
- Replaceability
- Adaptability

FUNCTIONAL SUITABILITY

- Functional correctness
- Functional completeness
- Functional appropriateness

MAINTAINABILITY

- Modularity
- Reuseability
- Modifiability
- Testability
- Analyzability

PERFORMANCE EFFICIENCY

- Capacity
- Resource utilization
- Time behavior

COMPATIBILITY

- Interoperability
- Co-existence

SECURITY

- Integrity
- Confidentiality
- Non-repudiation
- Accountability
- Authenticity

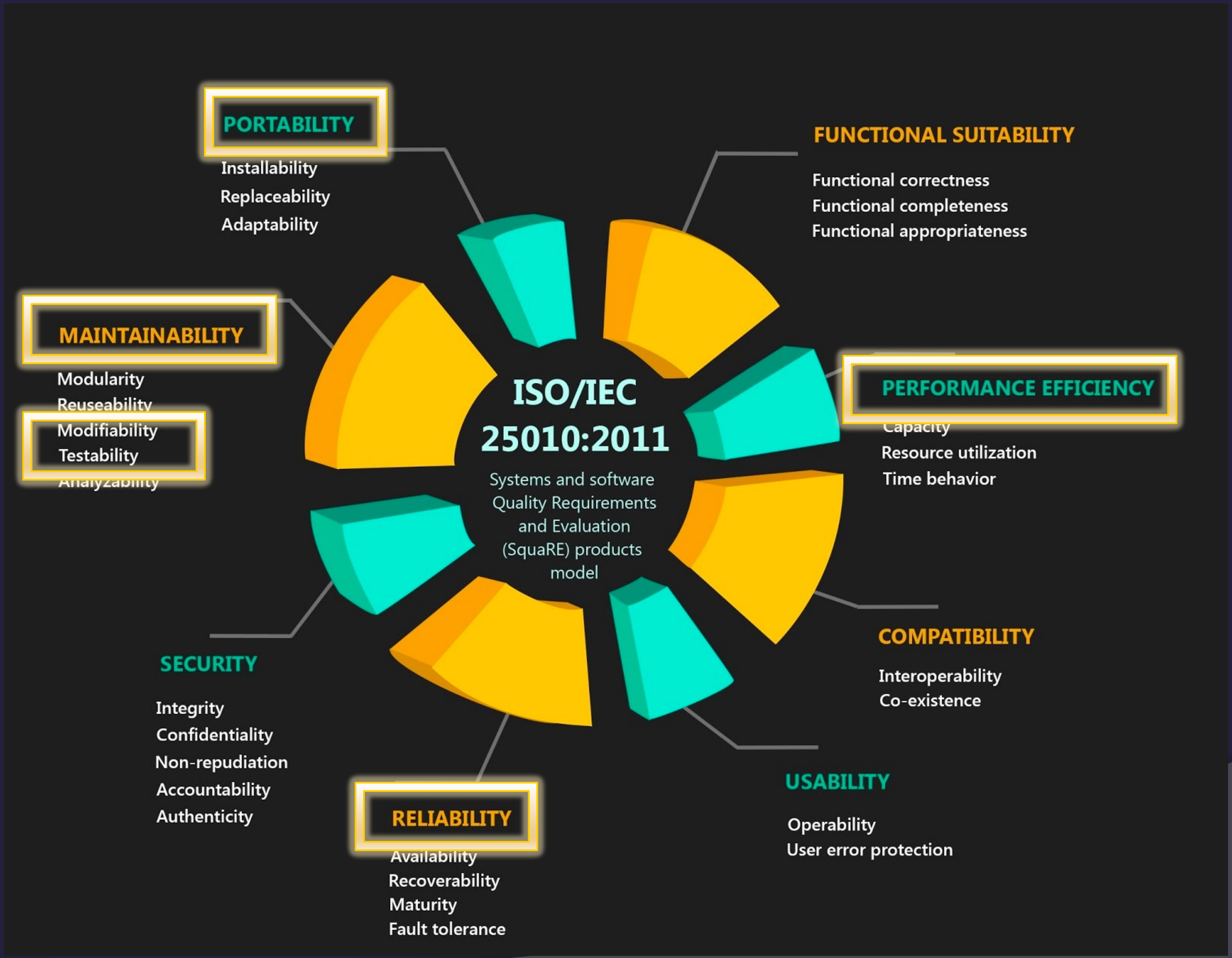
RELIABILITY

- Availability
- Recoverability
- Maturity
- Fault tolerance

USABILITY

- Operability
- User error protection

System Qualities – All stakeholders



ISO/IEC 25010:2011

Systems and software Quality Requirements and Evaluation (SquaRE) products model

PORTABILITY
- Installability
- Replaceability
- Adaptability

FUNCTIONAL SUITABILITY
- Functional correctness
- Functional completeness
- Functional appropriateness

MAINTAINABILITY
- Modularity
- Reuseability
- Modifiability
- Testability
- Analyzability

PERFORMANCE EFFICIENCY
- Capacity
- Resource utilization
- Time behavior

COMPATIBILITY
- Interoperability
- Co-existence

SECURITY
- Integrity
- Confidentiality
- Non-repudiation
- Accountability
- Authenticity

USABILITY
- Operability
- User error protection

RELIABILITY
- Availability
- Recoverability
- Maturity
- Fault tolerance

Can we say yes to everyone?

Very often the answer is NO

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*

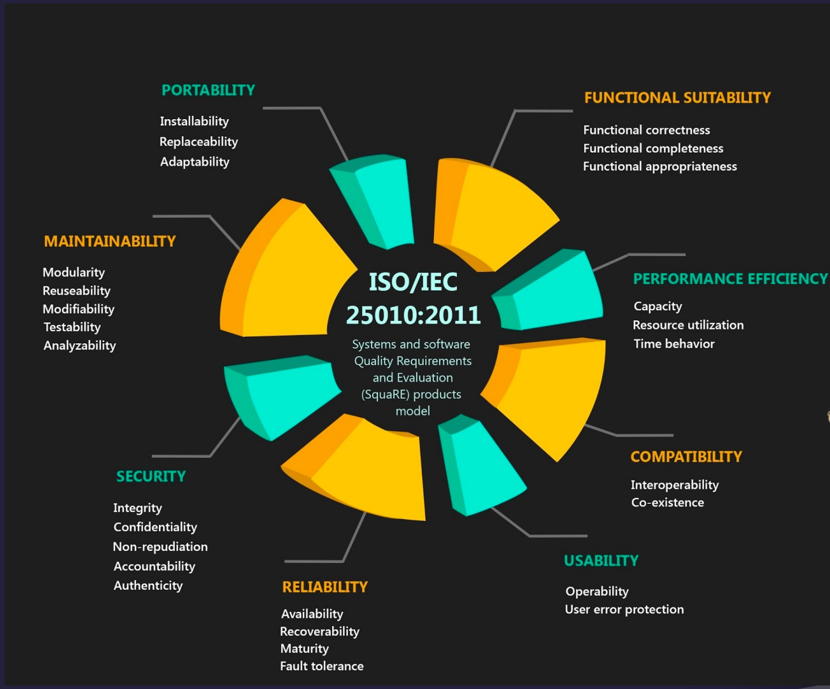
- Can we achieve both? We need to investigate more (e.g. with a workshop)

Can we say yes to both needs?

- We discuss the needs together with the stakeholders
  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

What’s the best architecture?

- *The best architecture is the best tradeoff among several qualities according to the business goals of the stakeholders*



ISO/IEC 25010:2011

Systems and software Quality Requirements and Evaluation (SquaRE) products model

PORTABILITY
Installability
Replaceability
Adaptability

FUNCTIONAL SUITABILITY
Functional correctness
Functional completeness
Functional appropriateness

MAINTAINABILITY
Modularity
Reuseability
Modifiability
Testability
Analyzability

PERFORMANCE EFFICIENCY
Capacity
Resource utilization
Time behavior

COMPATIBILITY
Interoperability
Co-existence

SECURITY
Integrity
Confidentiality
Non-repudiation
Accountability
Authenticity

RELIABILITY
Availability
Recoverability
Maturity
Fault tolerance

USABILITY
Operability
User error protection

Tradeoff(s)

- We generate solutions and scenarios
  1. Solution 1:
     - It takes 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

- Which one do we choose?
- Why?

Cost/Benefit and risk analysis

- Which solution is best?
  - Solution 1:
    - Waiting 2 more months (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 on the media
    - But we get customers from both platforms
  - 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

− risk

+ benefit

Scenarios and analysis

	Market share short-term	Market share long-term	Costs	Total
Solution 1	− − (we lose market share against competitor)	+ + (we deliver to both platforms) − (lack of visibility)	+ (cheaper in total)	
Solution 2	+ + (we gain market share against competitor)	+ (good visibility) − (no users in one of the platforms)	− (we need to rewrite)	

Scenarios and analysis

− risk

+ benefit

	Market share short-term	Market share long-term	Costs	Total
Solution 1	− − (we lose market share against competitor)	+ + (we deliver to both platforms) − (lack of visibility)	+ (cheaper in total)	0
Solution 2	+ + (we gain market share against competitor)	+ (good visibility) − (no users in one of the platforms)	− (we need to rewrite)	+1

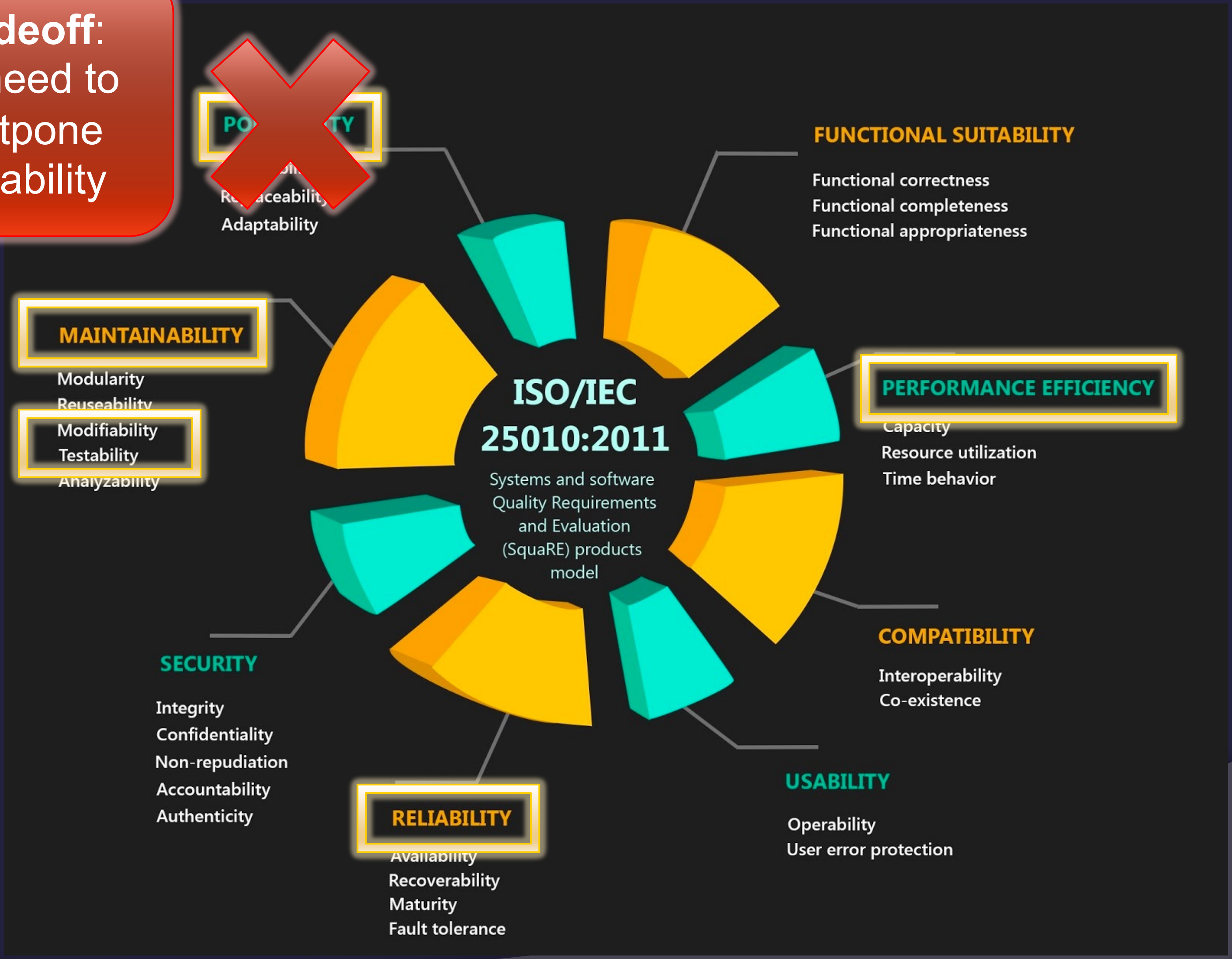
Tradeoff(s) example

- 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

System Qualities – Trade-off

Tradeoff: we need to postpone portability



PO TY

Re aceabilit

Adaptability

FUNCTIONAL SUITABILITY

Functional correctness
Functional completeness
Functional appropriateness

MAINTAINABILITY

Modularity
Reuseability
Modifiability
Testability
Analyzability

ISO/IEC 25010:2011

Systems and software Quality Requirements and Evaluation (SquaRE) products model

PERFORMANCE EFFICIENCY

Capacity
Resource utilization
Time behavior

COMPATIBILITY

Interoperability
Co-existence

SECURITY

Integrity
Confidentiality
Non-repudiation
Accountability
Authenticity

USABILITY

Operability
User error protection

RELIABILITY

Availability
Recoverability
Maturity
Fault tolerance

Available methodology

- ATAM
  - https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5177
- CBAM
  - https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513476

Principles of Software Architecture

Keeping down complexity

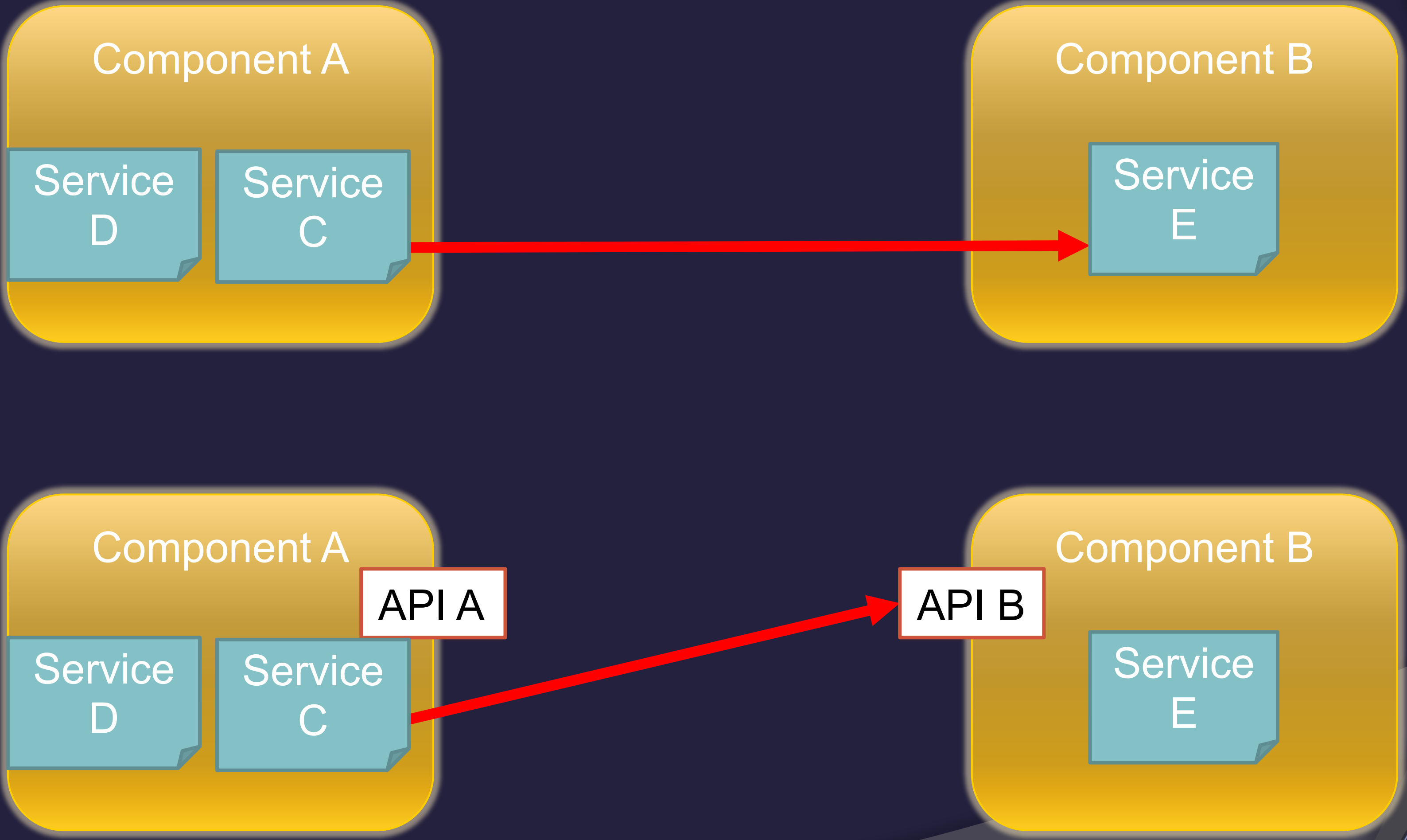
- We represent software with high level entities
  - Components, modules, layers, etc.
- And communication patterns
  - Interfaces, Dependencies, etc.
- To make something that is very complex understandable by humans
  - To share similar mental models

Component

Component A

- An element that implements a set of functionalities of features
- Examples:
  - Functional: a Graphical User Interface component
    - Where you define all the look and feel
  - Business-oriented: the Cart module
    - Implements where the user put the articles to buy

Components, services and APIs



API – Application Programming Interface

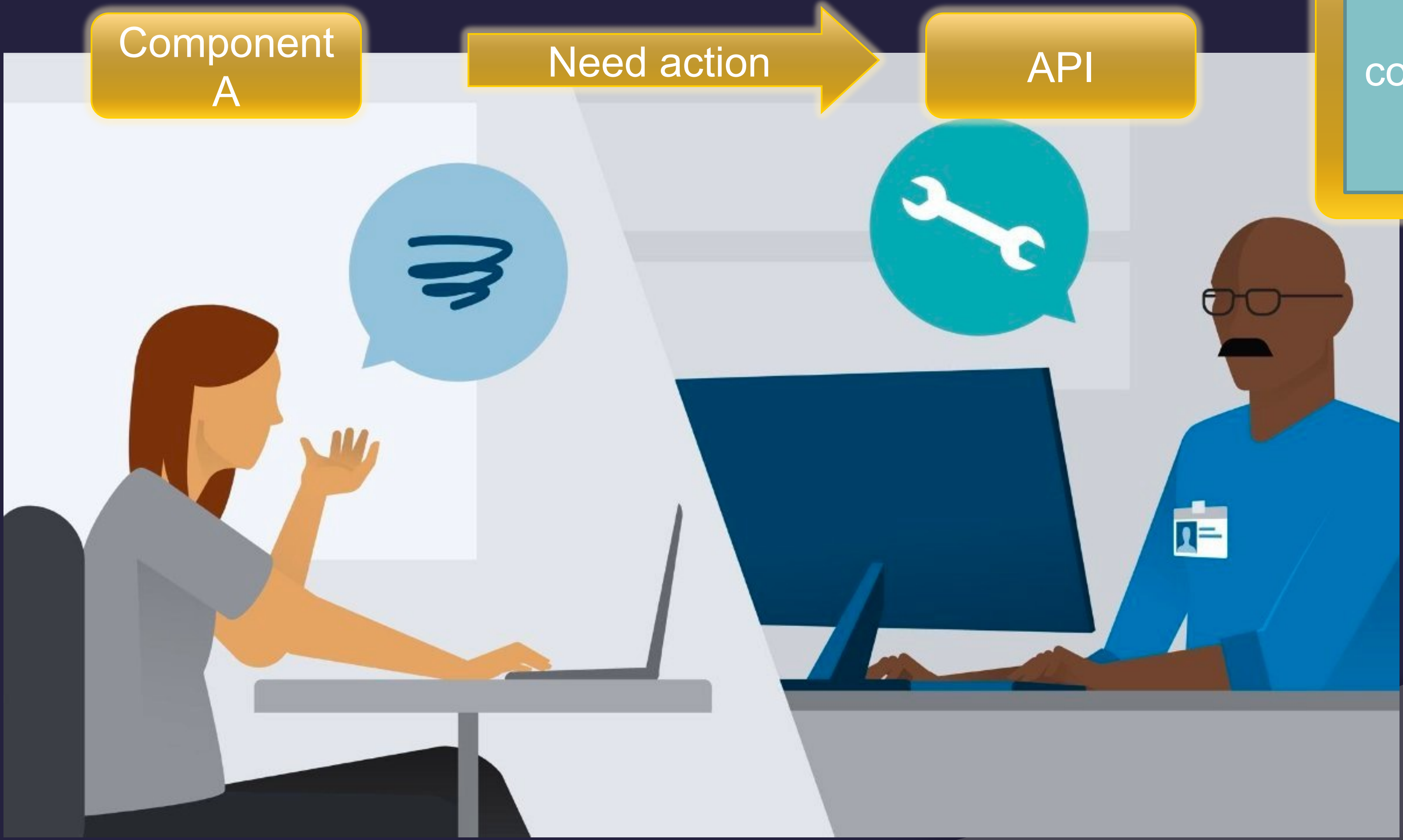
Component A

Need action

Component B

Very complicated code

API – Application Programming Interface



Component A

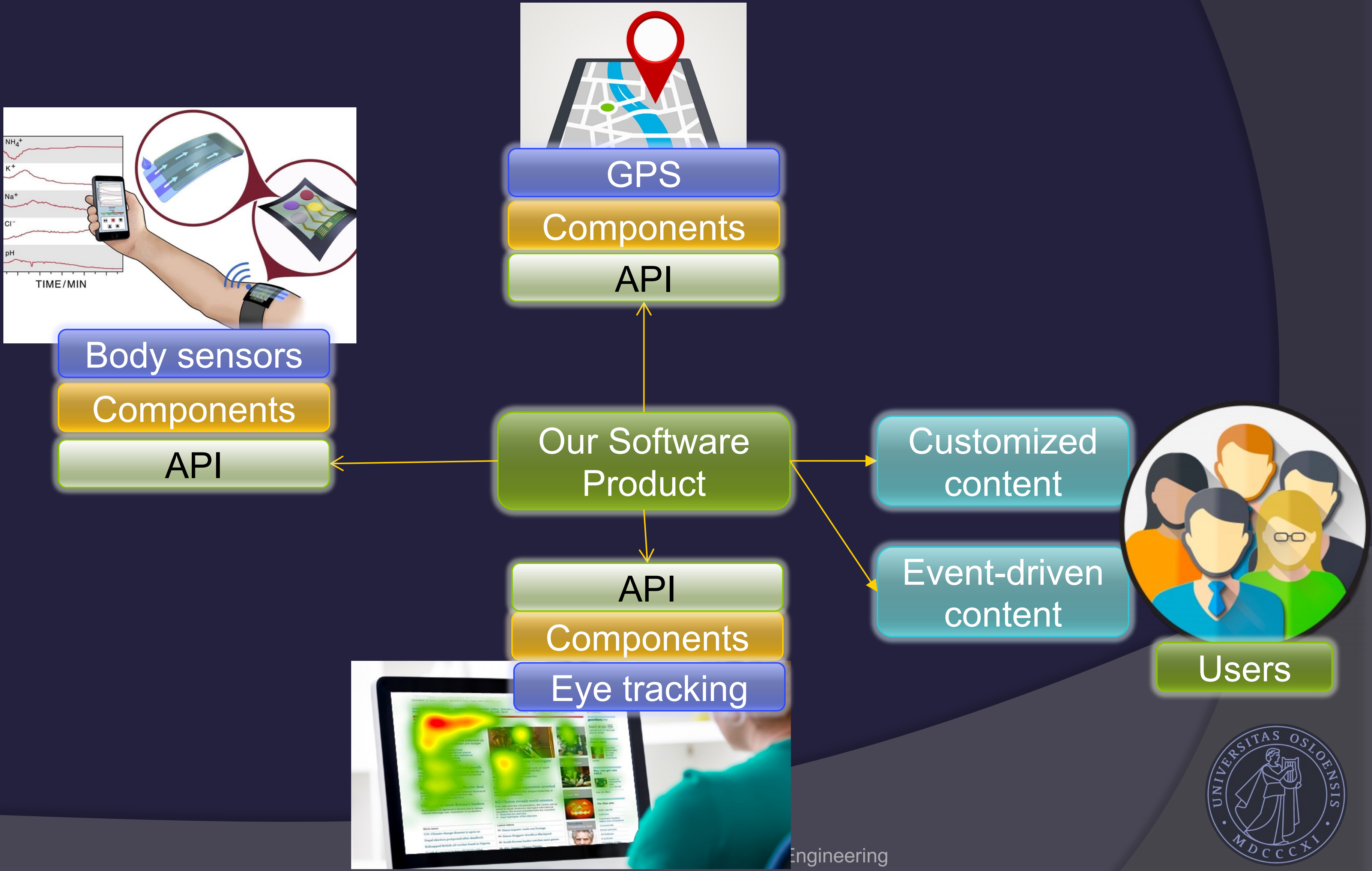
Need action

API

Component B

Very complicated code

External APIs



GPS

Components

API

Body sensors

Components

API

Our Software Product

Customized content

Event-driven content

Users

API

Components

Eye tracking

Beware of the terminology

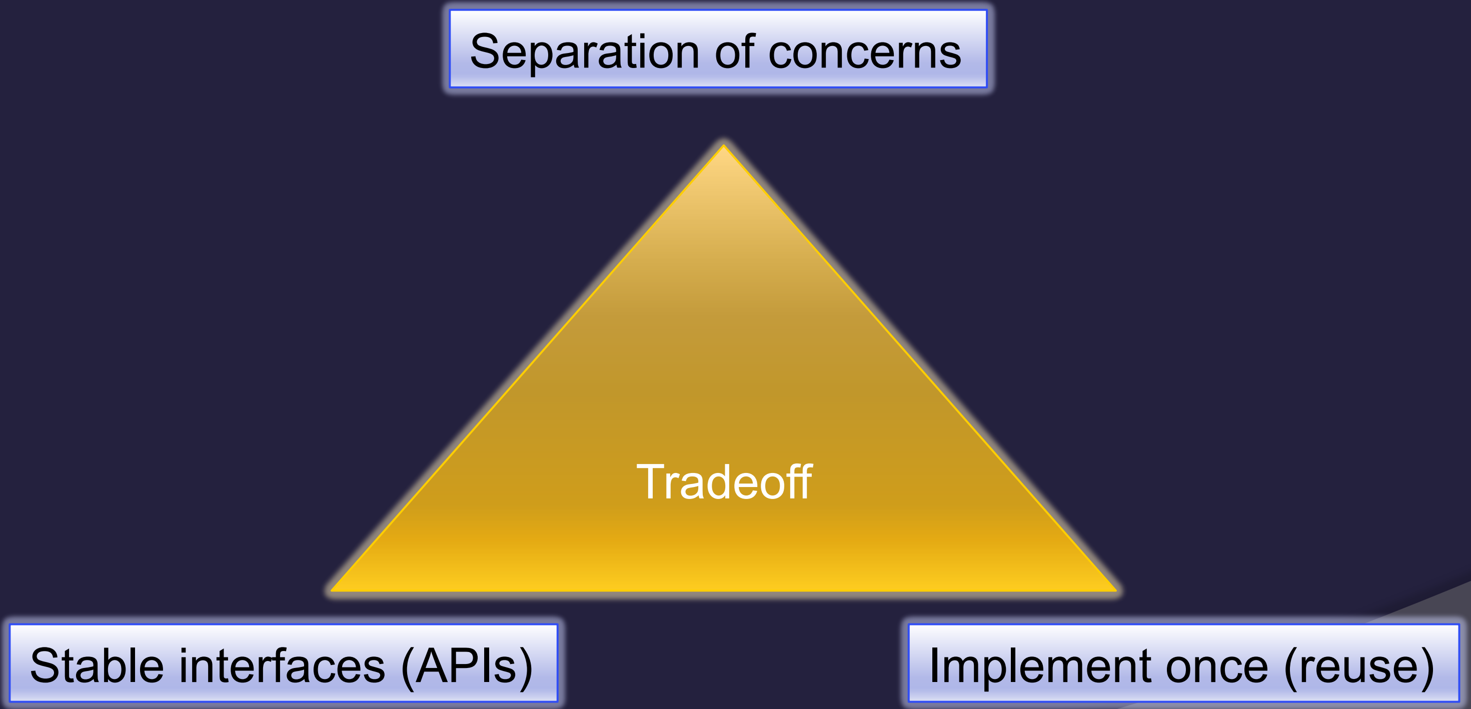
- Modules, components, services…

- … are often confused
- … are used in different ways in different contexts
- … are “just” containers

- Suggestion: try to understand from the context what they refer to

Architecture design

- Tradeoff to reduce complexity



Separation of concerns

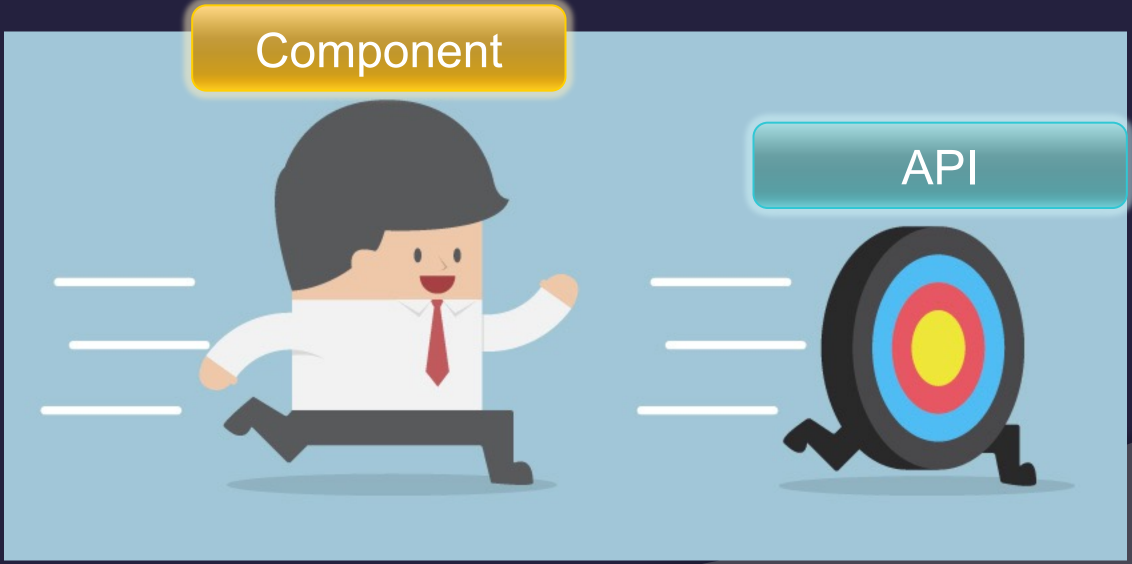
Tradeoff

Stable interfaces (APIs)

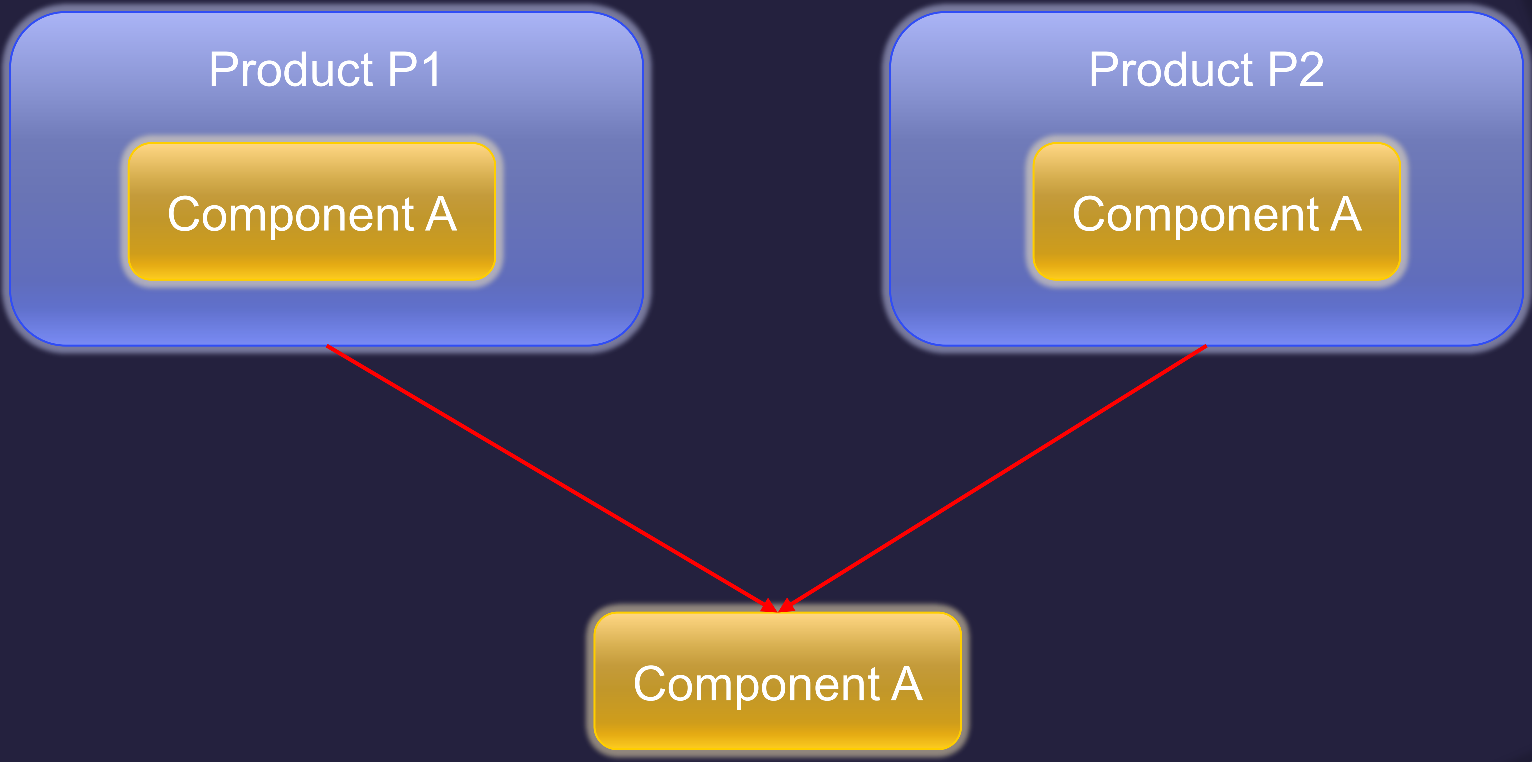
Implement once (reuse)

1. Why stable APIs?

- If the API changes continuously of my component
- All the other components need to change with me!!

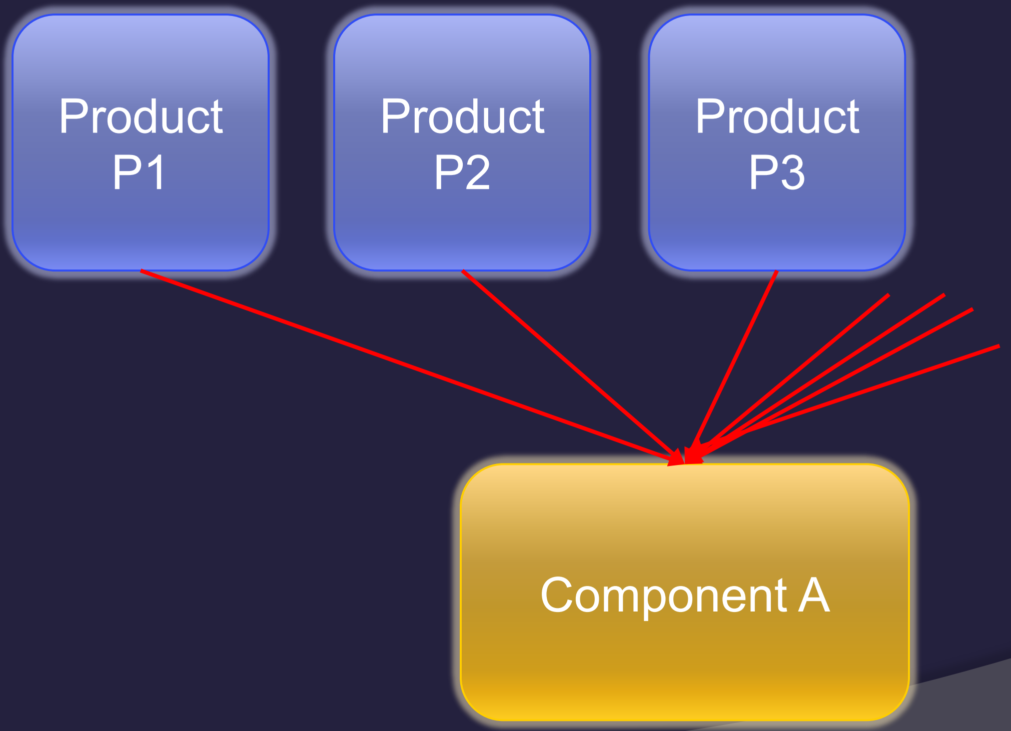


2. Reuse



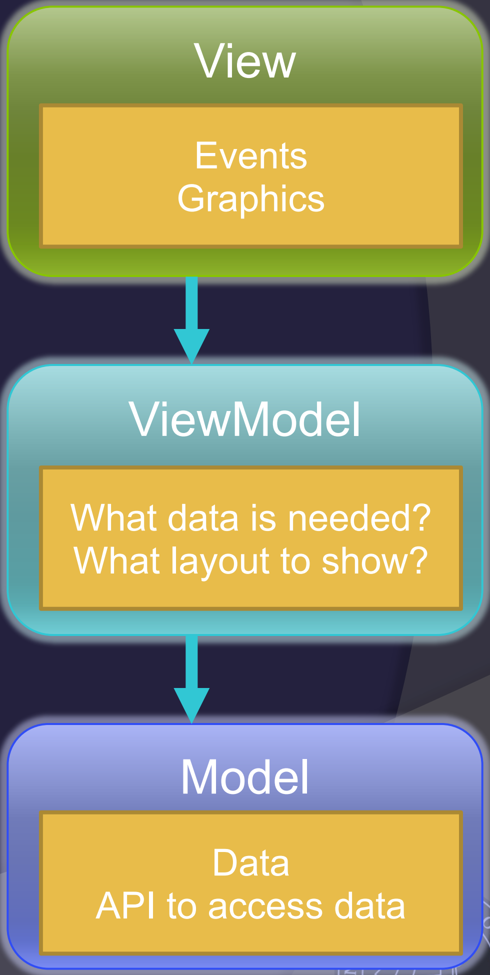
What’s the problem with too much reuse?

- Too many stakeholders!
- Too much coordination!

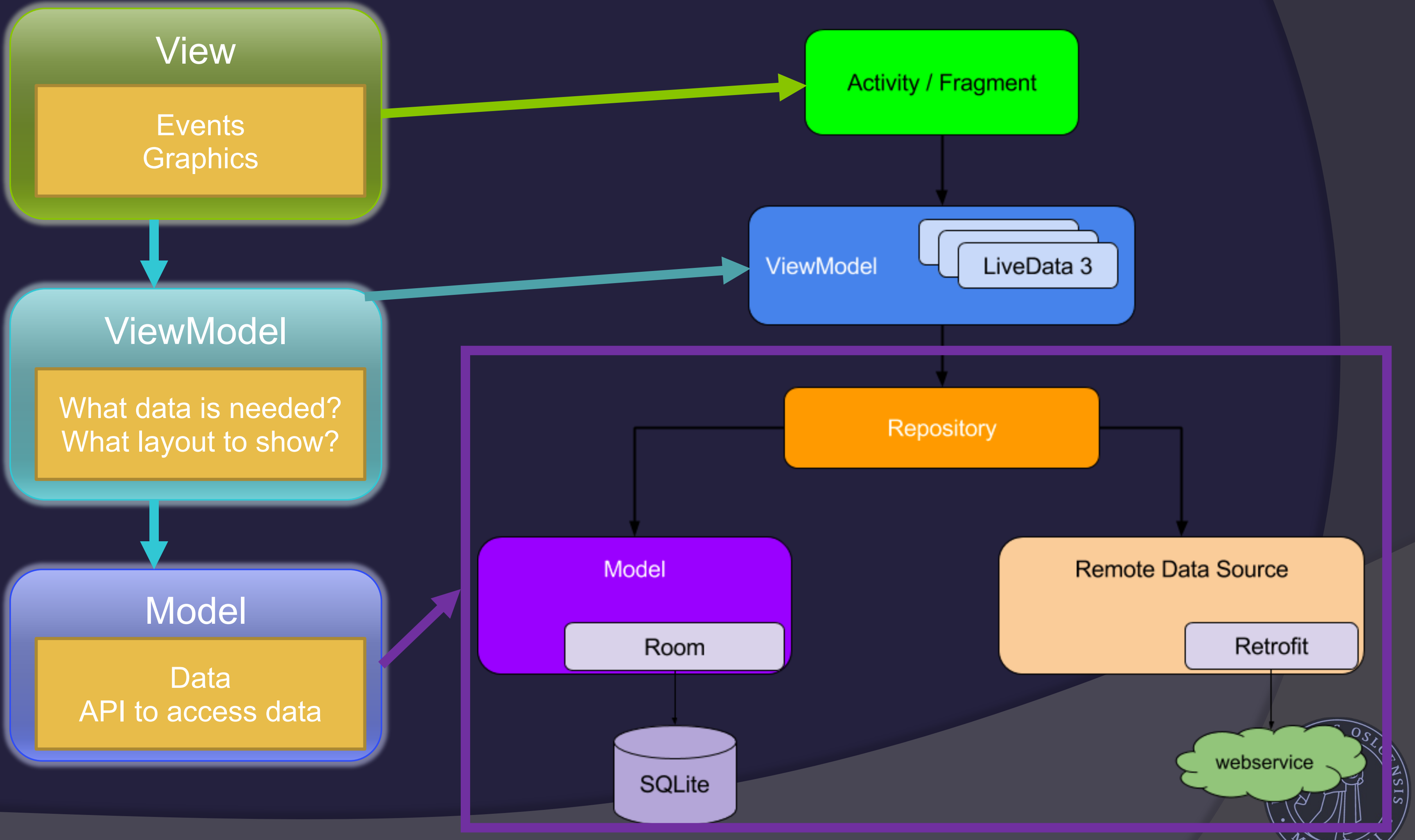


3. Good separation of concerns

- In Android, the following 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"
  - **ViewModel**:
    - interprets the user events and what data is needed
    - chooses the right way to show the results



MVVM in Android

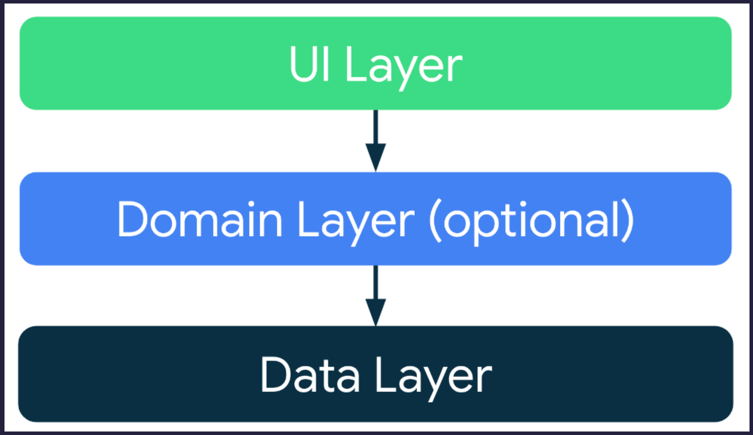


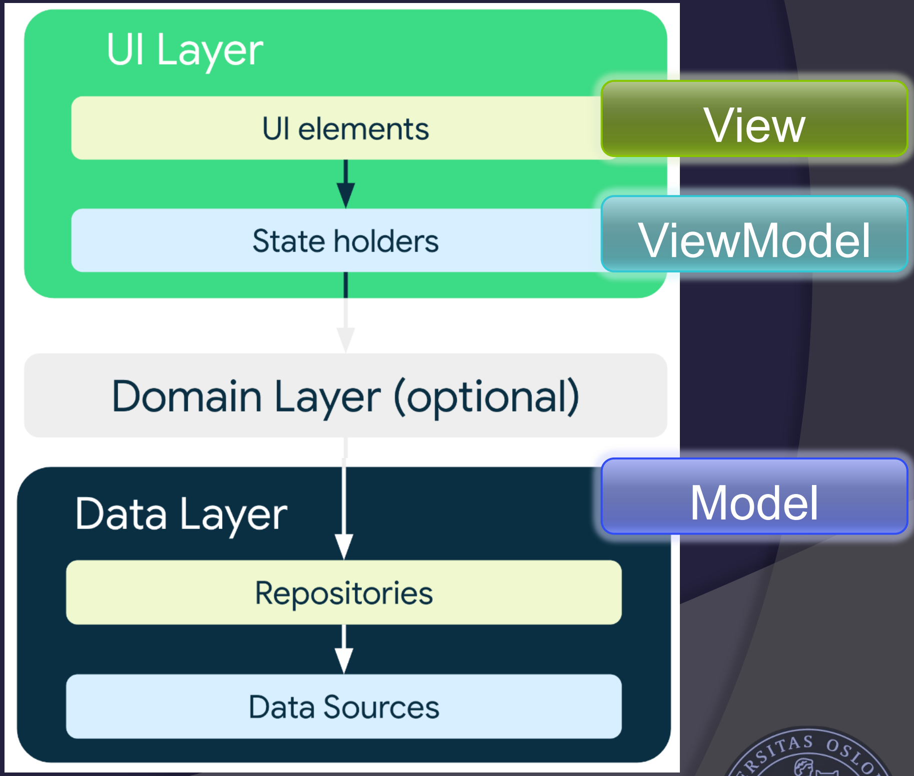
Architecture in Android

- Architecture guidelines in Android
  - https://developer.android.com/topic/architecture

Updated architecture for Android

- Essentially the same concepts





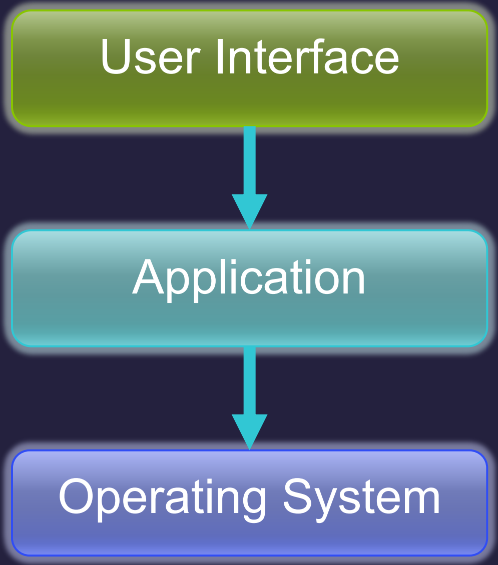
View

ViewModel

Model

Layers

- High level separation of concerns
- A way to reduce dependencies (only one way)



Other architectural styles

- Microservices
- Client-server
- Cloud
- …

- More in other courses (e.g. IN5140)

Technical Debt

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

APPROVED

- In two words:
  - Technical Debt

What is Technical Debt?

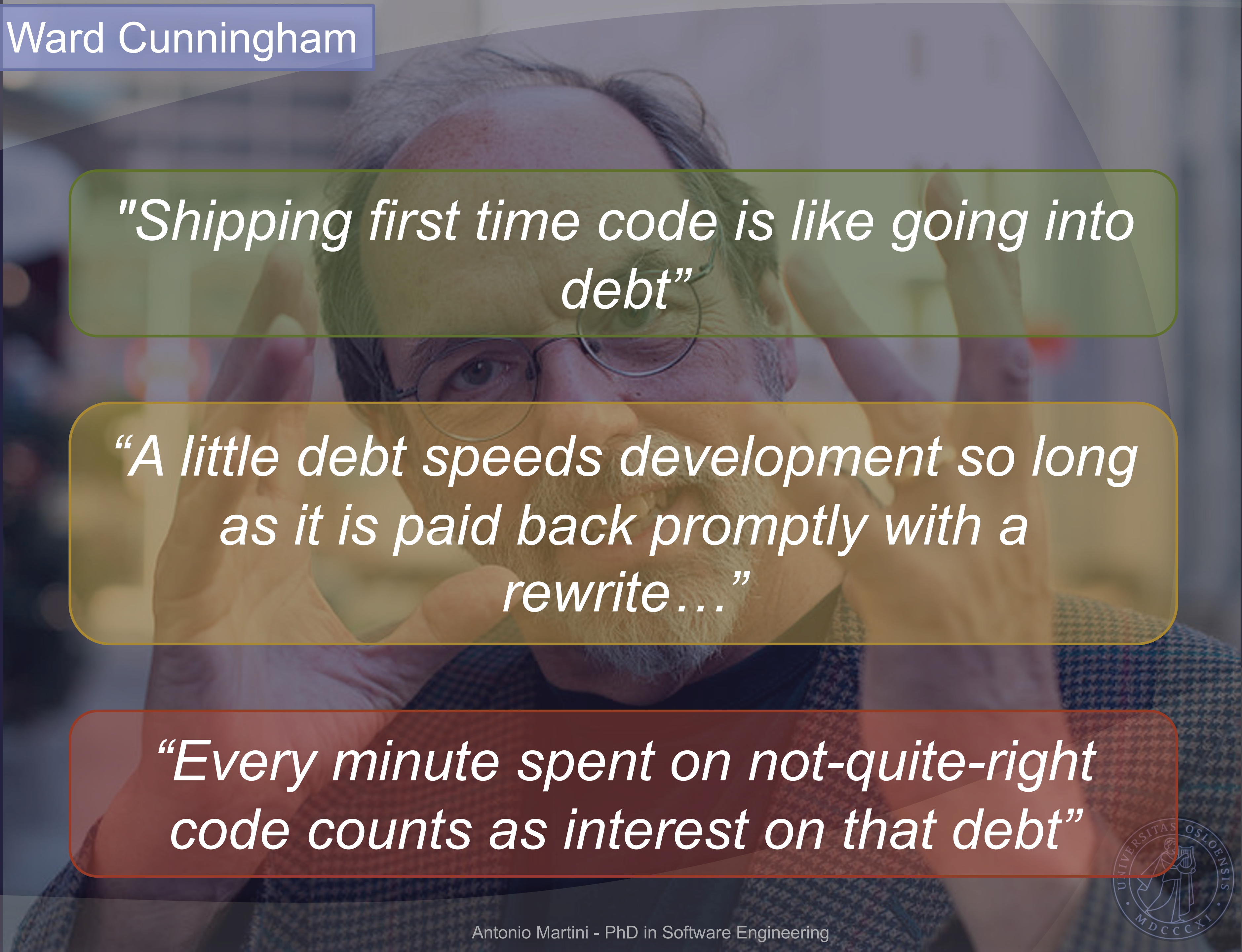
What the users see

What the developers see

What’s the problem? It works!...

- …for now…
- It might have leakages
  - Every now and then, the water doesn’t flow
- It costs a lot to maintain
  - Every time the plumber tries to fix it it takes days!
- It’s hard to extend
  - Forget about connecting a washing machine!

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*

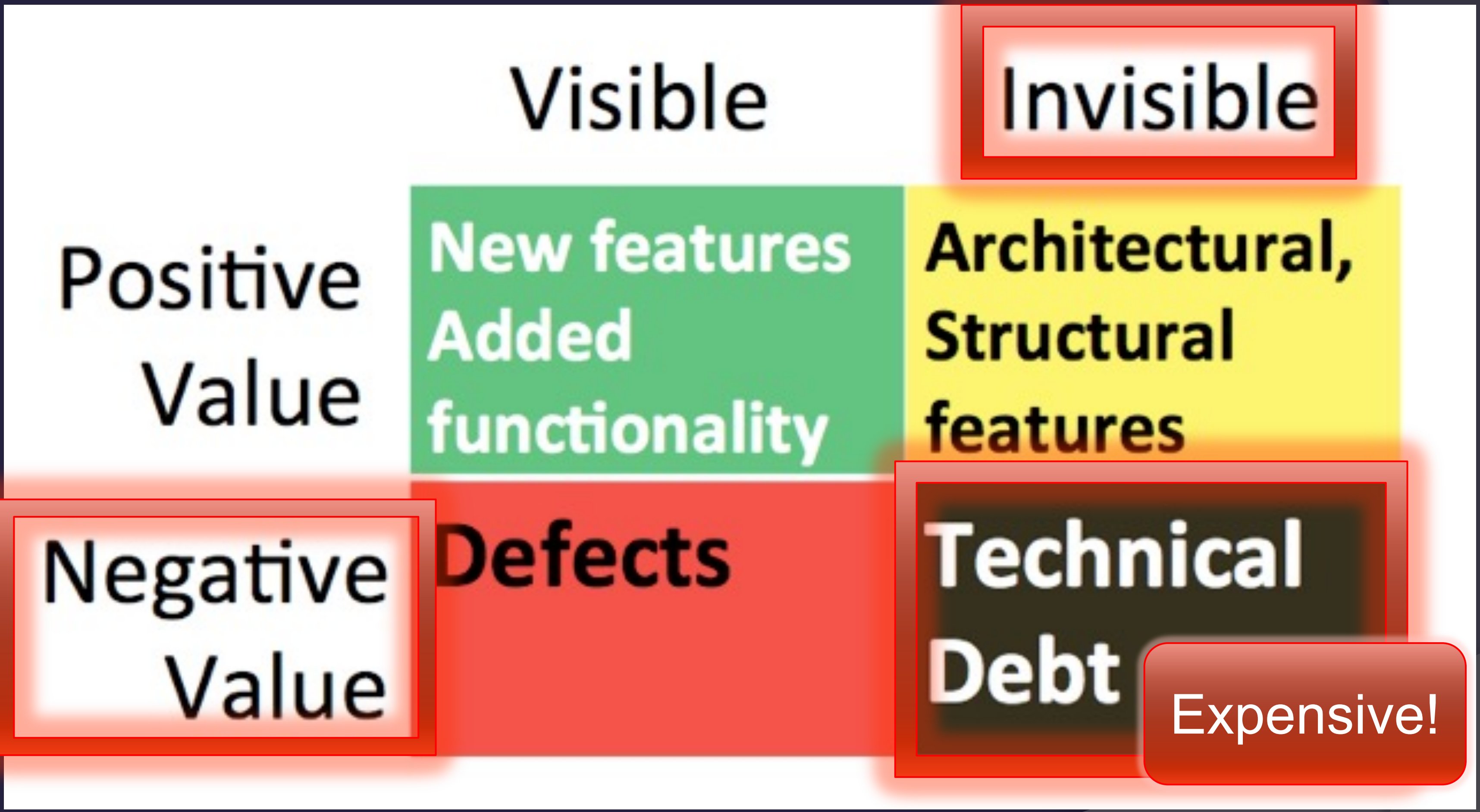
P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, “Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)

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)

Technical Debt and software development

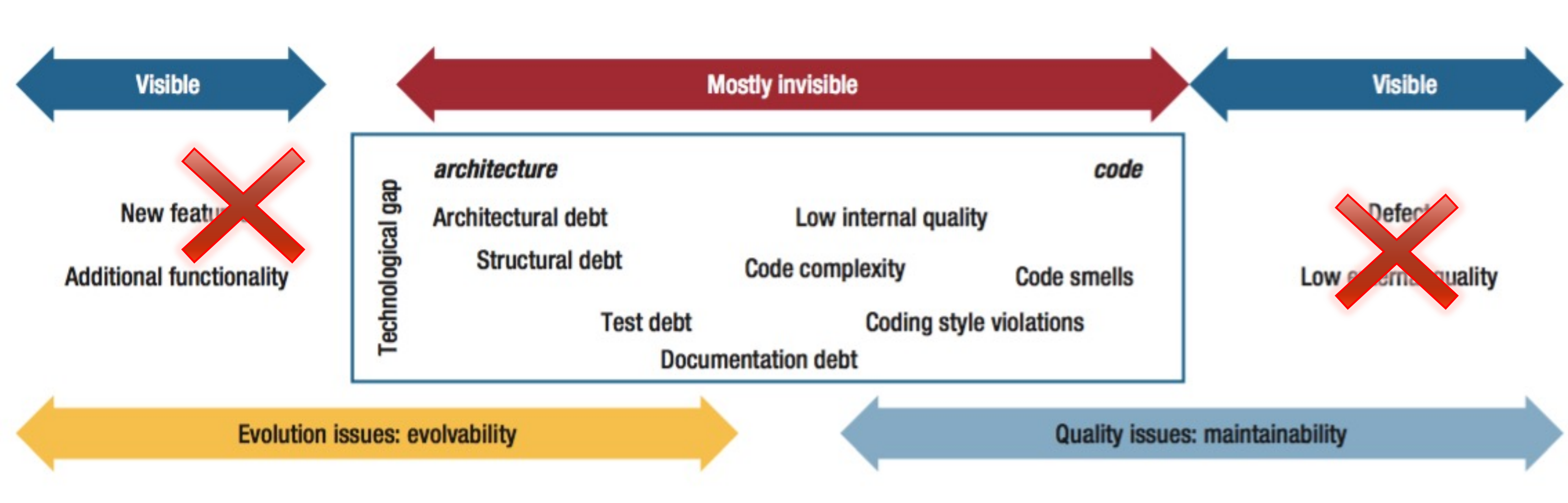


	Visible	Invisible
Positive Value	New features Added functionality	Architectural, Structural features
Negative Value	Defects	Technical Debt

Expensive!

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

The TD landscape of kinds of TD



Visible

Mostly invisible

Visible

New featu

Additional functionality

Technological gap

architecture

code

Architectural debt

Low internal quality

Structural debt

Code complexity

Code smells

Test debt

Coding style violations

Documentation debt

Defect

Low e erna uality

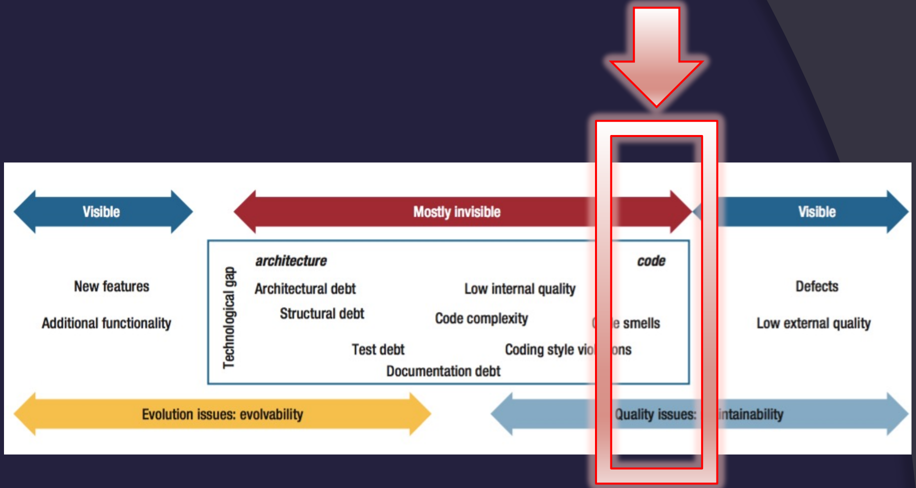
Evolution issues: evolvability

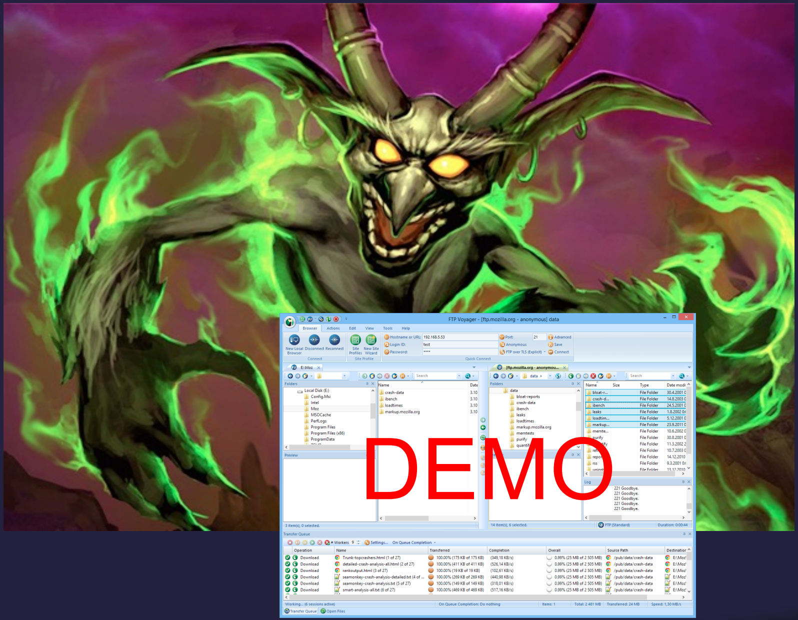
Quality issues: maintainability

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

Example

- Demo
  - “Magic” Numbers
    - It’s Code debt





Beware of the dreadful

Demo-Demon

What was technical debt in this example?

- Debt
  - Sub-optimal solution
- Principal
  - Cost of repaying (or not taking) the debt
- Interest
  - Cost of impact
- Was it worth taking the debt?

Example 1

- We use changes as cost
- We want to change the deck size from 40 to 52

- Debt
  - Sub-optimal solution
    - Not using a constant for the deck size

- Principal
  - Cost of repaying (or not taking) the debt
    - Implementing the constant in the beginning:
      - +1 change

- Interest
  - Cost of maintenance (or other impacts)
    - When we changed the deck size
      - +5 changes

- Was it worth taking the debt?
  - Principle / interest = 1/5
  - We would have saved 4 changes (4/5)

Example 2

- We use changes as cost
- We add a method and we want to change the deck size as in Example 1

- Debt
  - Sub-optimal solution
    - Not using a constant for the deck size

- Principal
  - Cost of repaying (or not taking) the debt
    - Implementing the constant in the beginning:
      - +1 change

- Interest
  - Cost of maintenance (or other impacts)
    - When we changed the deck size
      - +6 changes

- Was it worth taking the debt?
  - Principle / interest = 1/6
  - We would have saved 5 changes (5/6)

As the software grows, the interest also grows!

Example 3

- We use changes as cost
- See example 2, but this time we run the program

- Debt
  - Sub-optimal solution
    - Not using a constant for the deck size

- Principal
  - Cost of repaying (or not taking) the debt
    - Implementing the constant in the beginning:
      - +1 change

- Interest
  - Cost of maintenance (or other impacts)
    - When we changed the deck size
      - +6 changes
    - When we run the script
      - There is a bug

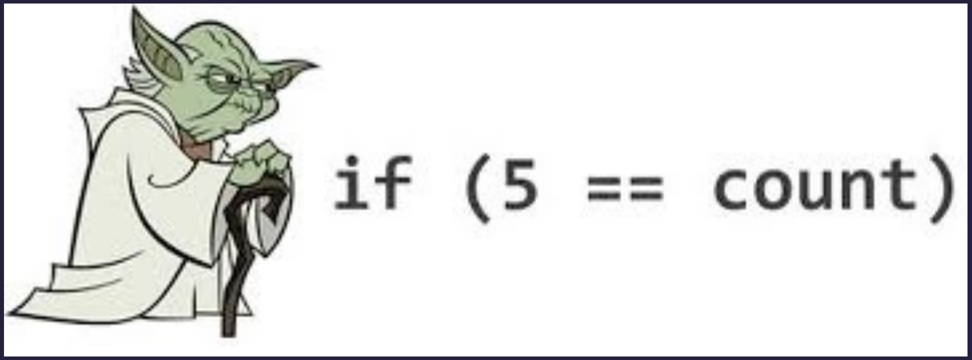
It's not only about cost
It's also a risk!

- Was it worth taking the debt?
  - Same as Example 2 but there was also the risk of bugs

Suggesting refactoring

- During the project, we need to refactor
  - E.g. Removing technical debt
- In your project, you will get the opportunity to refactor one or more files during one of the activities...
- ...using AI

Another (funny) example of Code debt

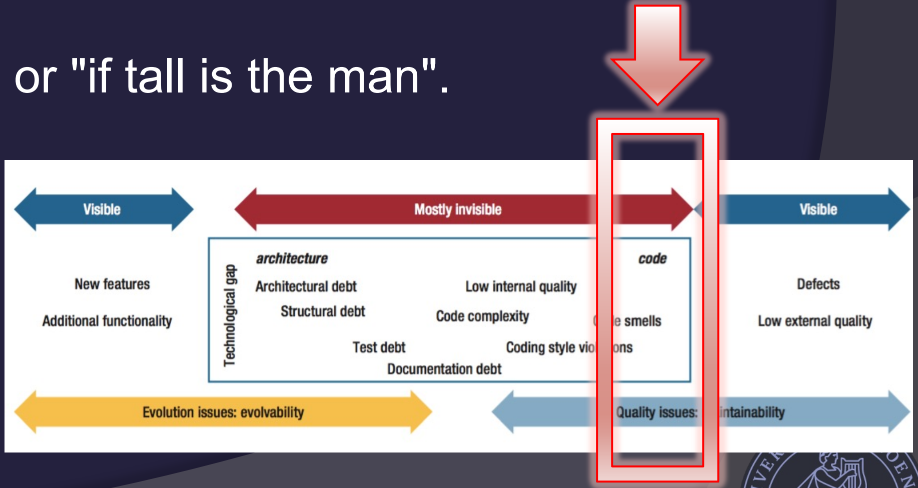


```
if (5 == count)
```

Yoda Condition*

Using if(constant == variable) instead of if(variable == constant), like if(4 == foo).

Because it's like saying "if blue is the sky" or "if tall is the man".



Visible

Mostly invisible

Visible

New features

Additional functionality

Technological gap

architecture

Architectural debt

Structural debt

Test debt

Documentation debt

Low internal quality

Code complexity

Coding style vio ons

code

e smells

Defects

Low external quality

Evolution issues: evolvability

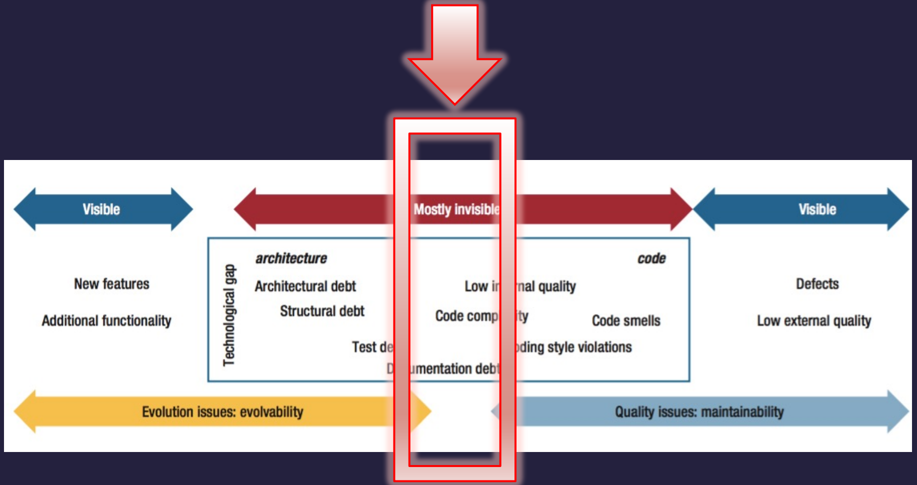
Quality issues: ntainability

* www.dodgycoder.net/2011/11/yoda-conditions-pokemon-exception.html

Example of Documentation debt

Ninja Comments*

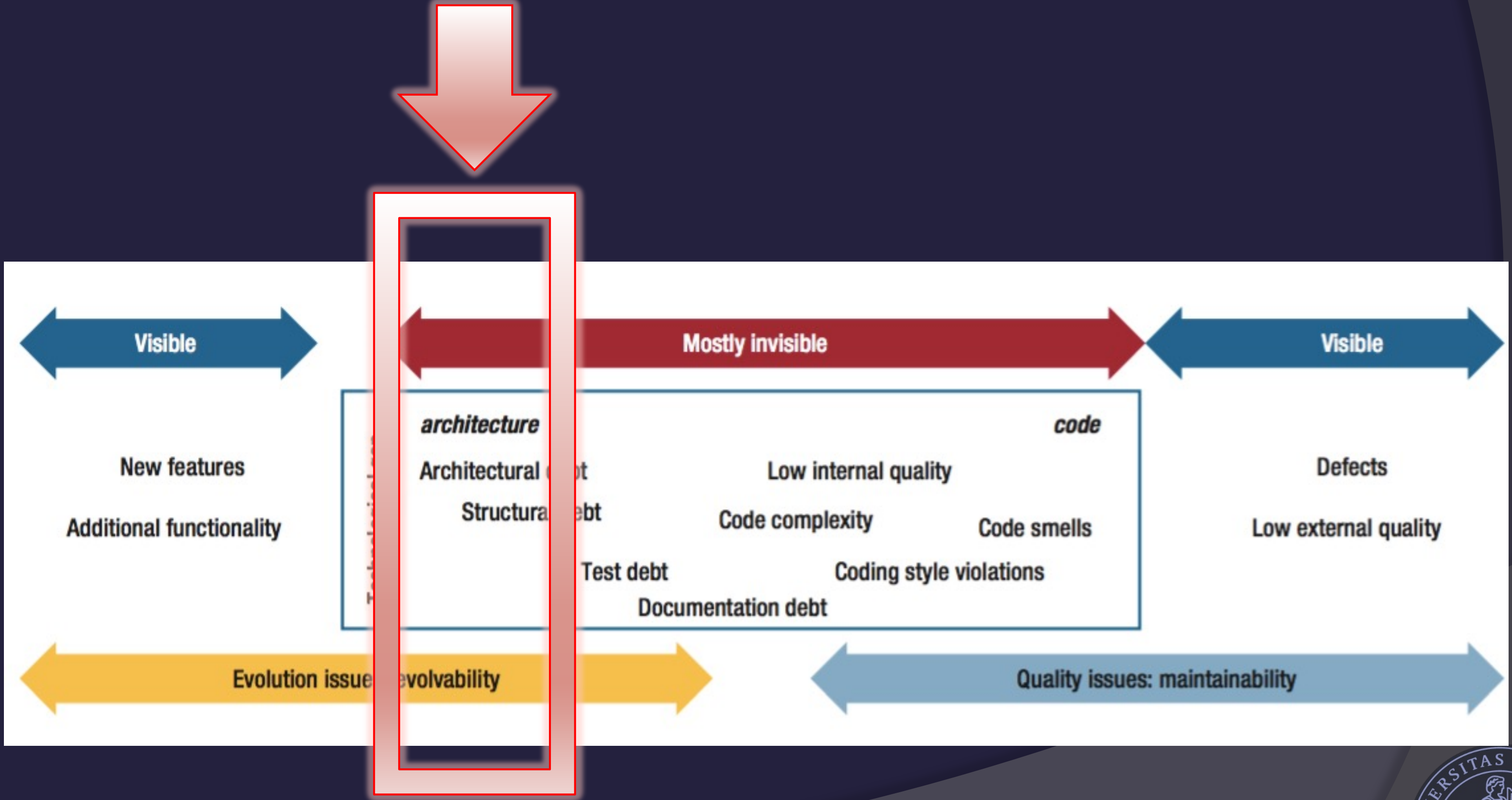
Also known as invisible comments, secret comments, or no comments.



* www.dodgycoder.net/2011/11/yoda-conditions-pokemon-exception.html

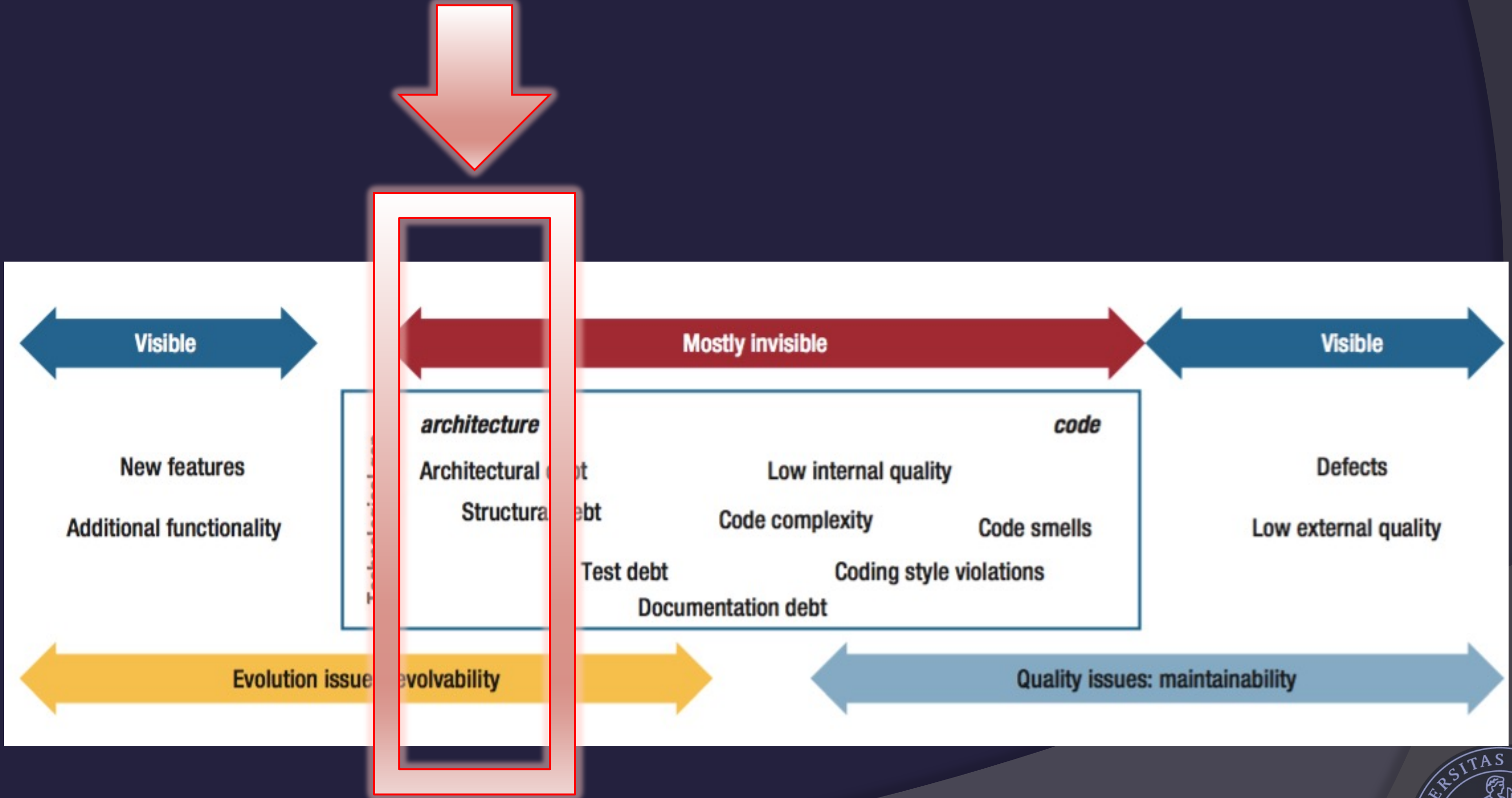
Horror Story

- Technical debt and Architecture



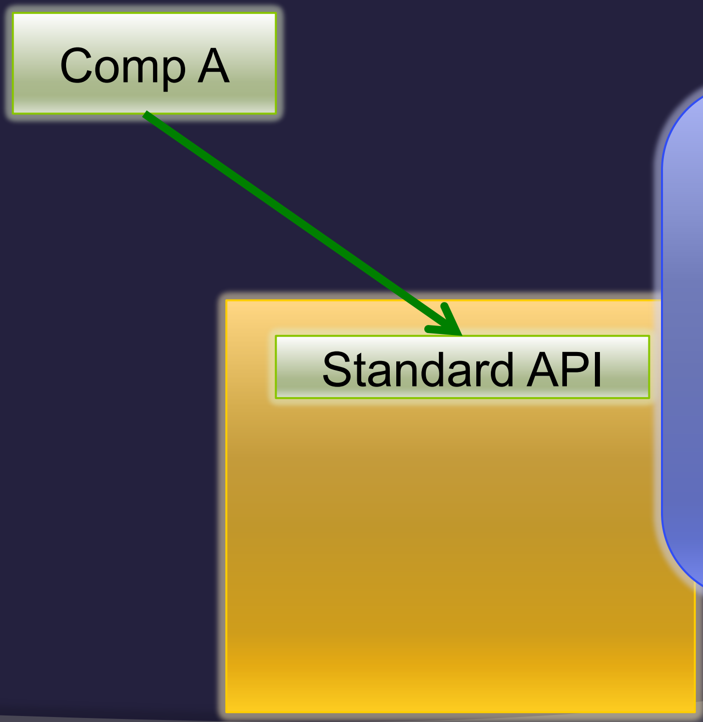
Horror Story

- Technical debt and Architecture



Optimal architectural decision

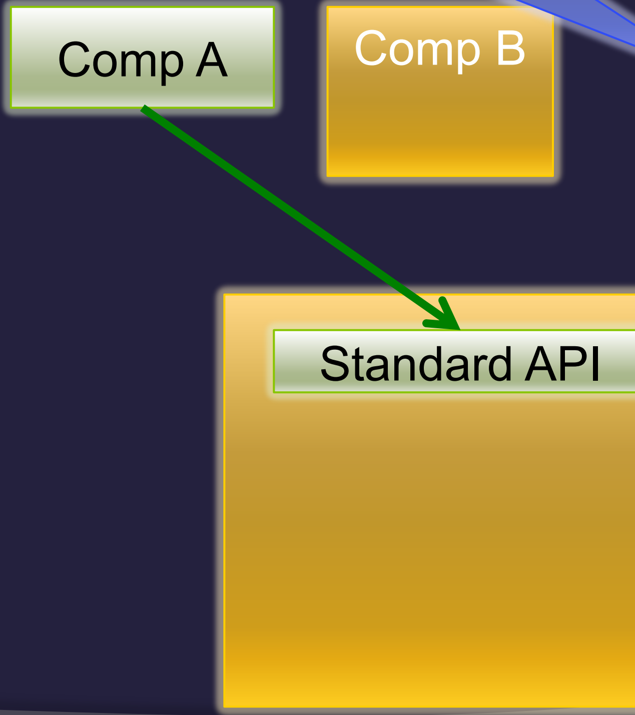
- Example:
  - Standard public API



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

During feature development…

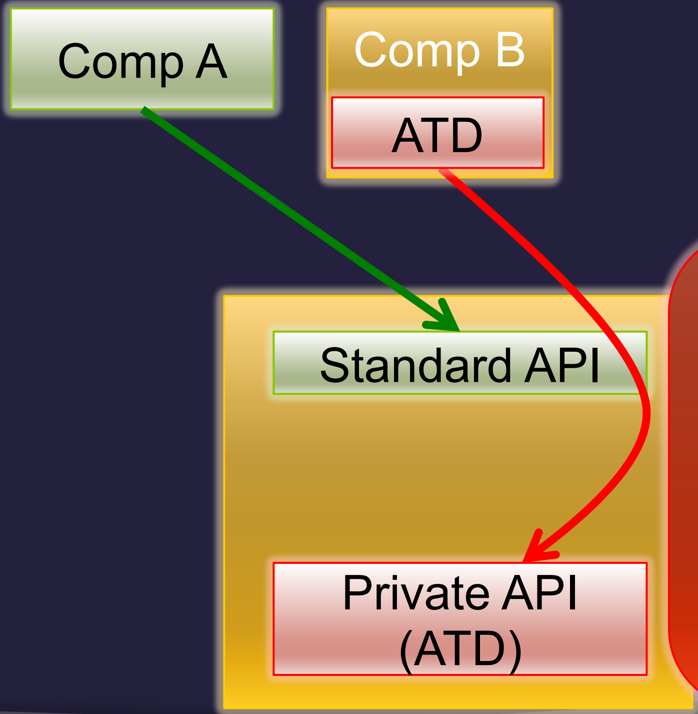
No problem, let’s add a component B. The teams will use the standard API!



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

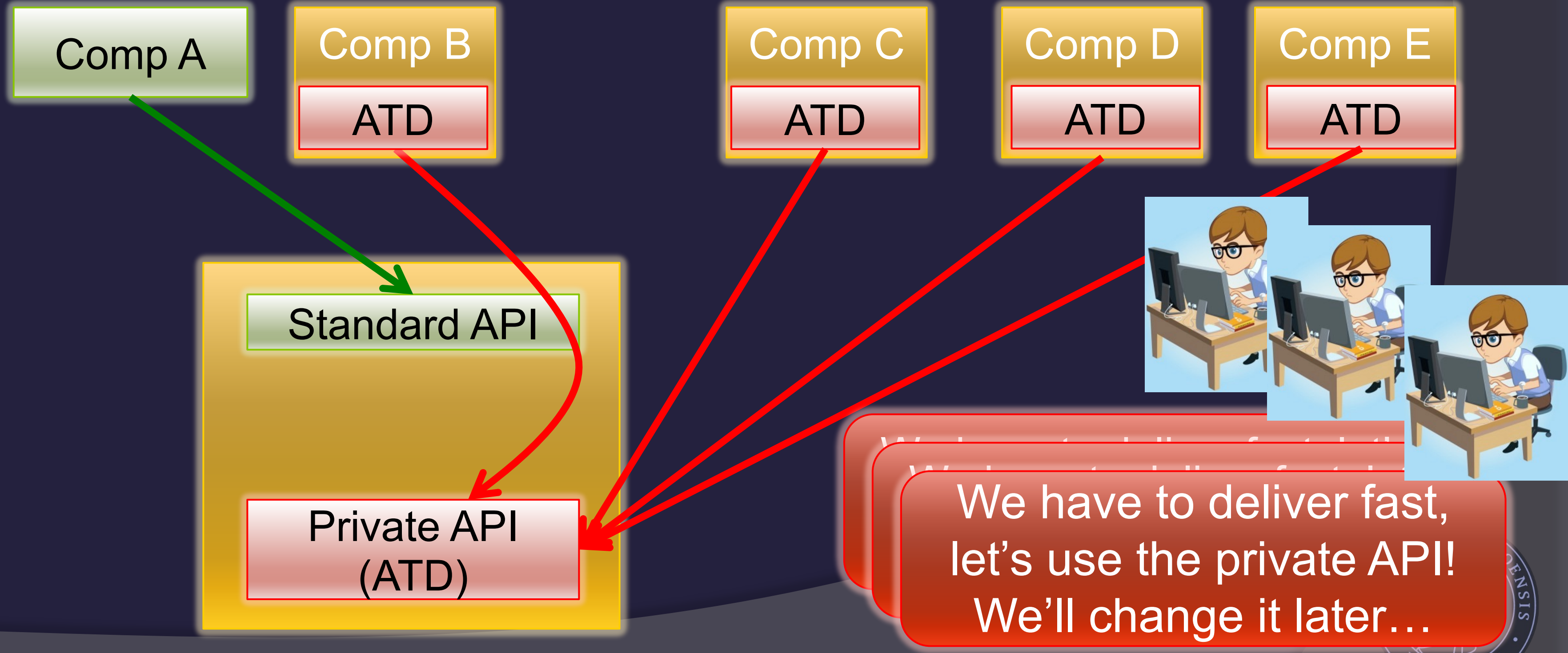
We need these new features! Our competitor is already delivering them!

Fast!

…the accumulation of sub-optimal decisions…

Fast!

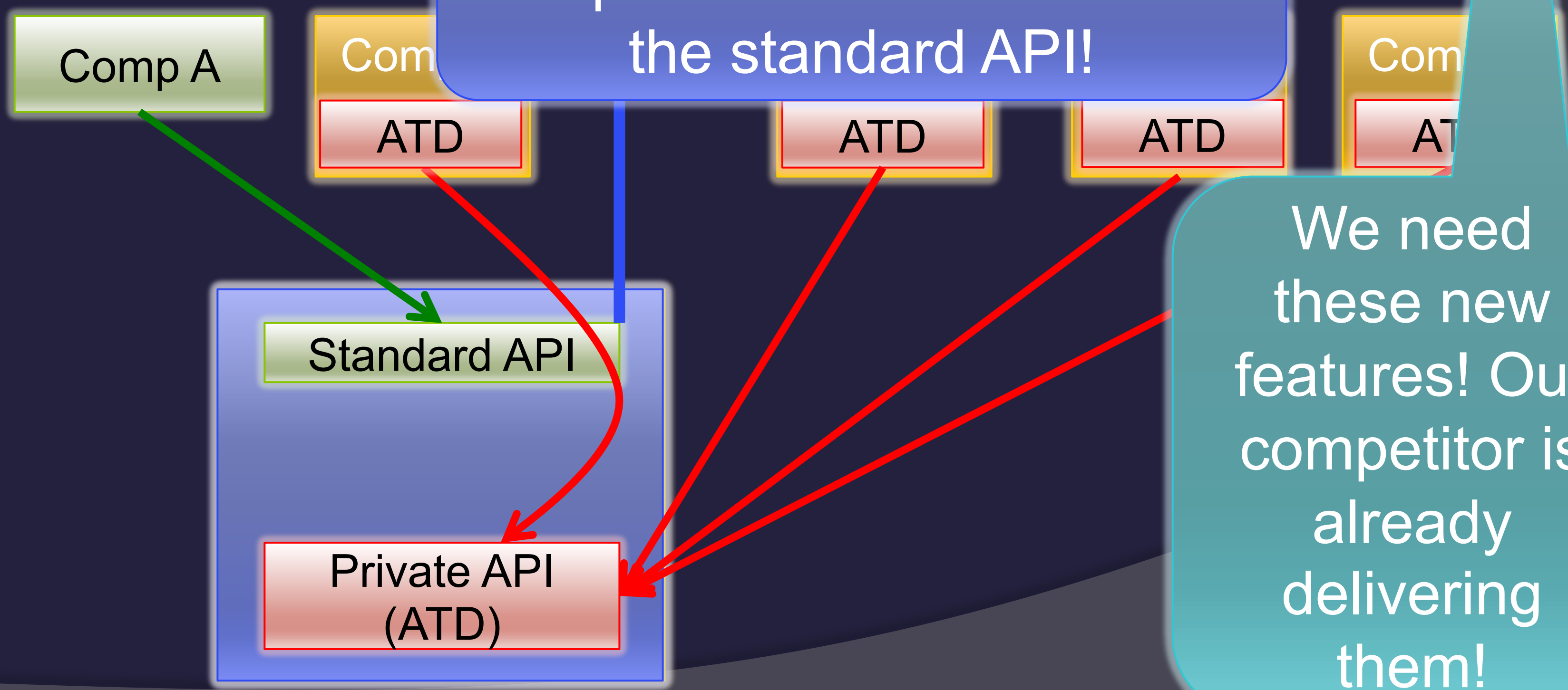
- The violation is spreading to many components



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

...until, one day...

- New requirement

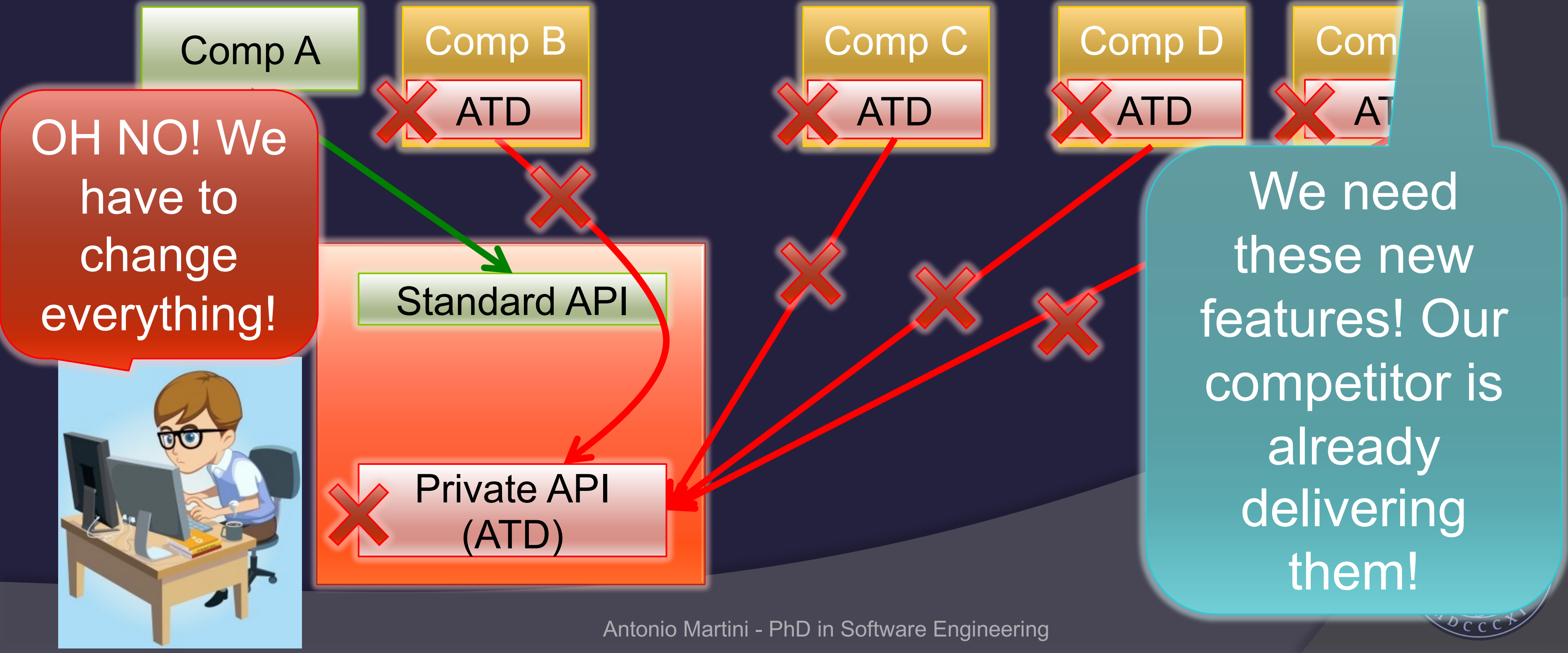


Ok, we can replace this component. The teams used the standard API!

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*



Comp A

Comp B

ATD

Comp C

ATD

Comp D

ATD

Com

AT

Standard API

Private API (ATD)

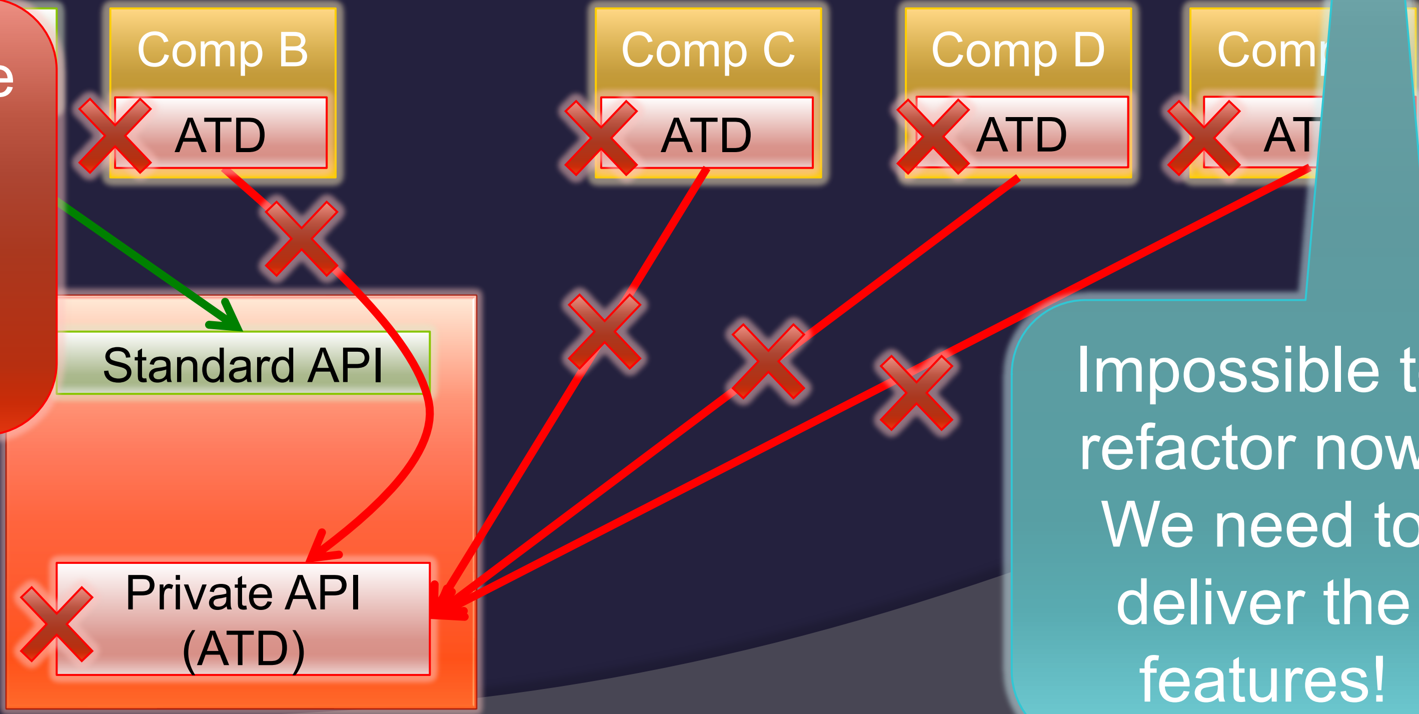
OH NO! We have to change everything!

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

...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 to sum up, what's important about Software Architecture?

Summary on Software Architecture

All that is important and costly to change later

Architecture is about tradeoffs and communication

Architecture is design should reduce complexity

The wrong tradeoffs create dangerous technical debt

Questions?

Comments?

- antonio.martini@ifi.uio.no