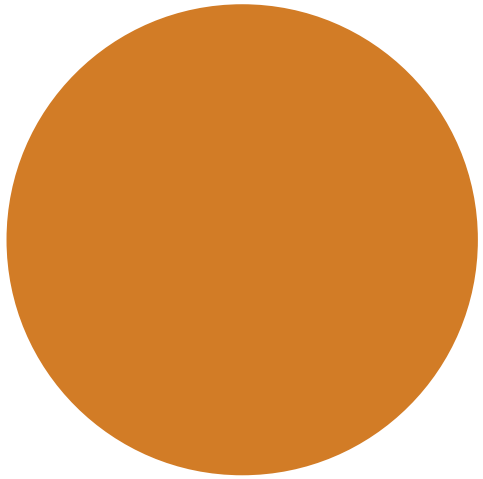
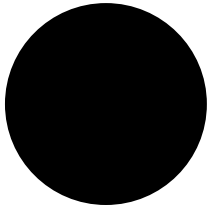




Enterprise Application Integration

Lesson 3
EAI Introduction

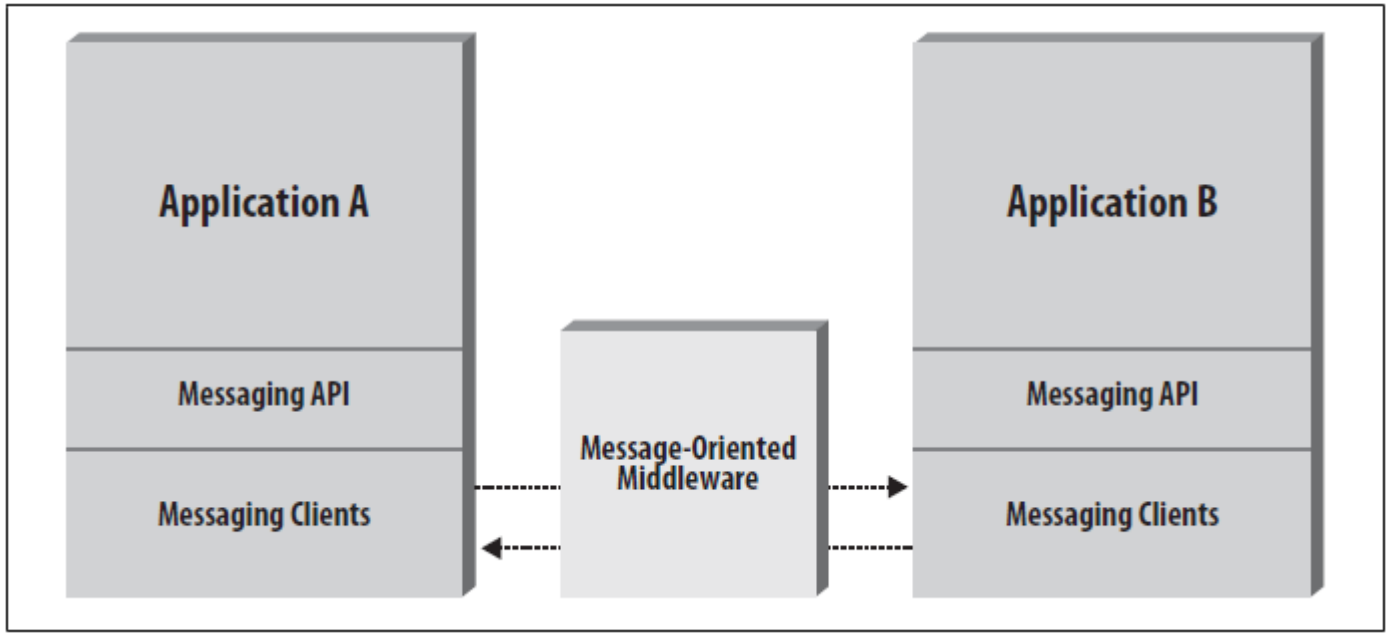
Hanzehogeschool Groningen
University of Applied Sciences

Key - EAI related – Business Software Architecture questions

- What are the EAI quality requirements for software components and integration?
- How to structure the IT landscape in components?
- Which integration styles and technologies should we use?
- Do we need middleware software from suppliers to serve as an integration platform?

Middleware

- Software which is specifically designed to integrate Business Applications is called: Middleware. This is a synonym for Integration Platform.
- When the only integration style of the middleware is Messaging then it is: Message Oriented Middleware (MoM).
- Hybrid Integration Platform = Middleware which supports more integration styles than just messaging.
- Middleware integrates heterogeneous applications.



Benefits of good components

1. Development without or minimal side effects and coordination with other teams.
2. Understand individual components independent of the rest.
3. Test components in isolation.
4. Easy replacement.

An indicator for a good architecture:

A software problem is limited to 1 component.

Source: Dowalil 2018

Easy replacement is King!

- Easy replacement of modules is a key design goal!
- More dependencies between components

  => harder to replace,
  => harder to fix system failures

EAI quality requirements for software components

Software Design Principles with high relevance for the Macro Architecture

High cohesion	Loosely coupled	Separation of concerns	Information hiding principle / Encapsulation
Hierarchical structure	Open closed principle	Keep it simple	Don't repeat yourself

Loosely coupled principle & Enterprise Applications

- "Integrated applications should minimize their dependencies on each other so that each can evolve without causing problems to the others."
- Some common dependencies:
  - Different data formats which require data format transformation
  - Timeliness processing
    - => shared data should be processed by the other application in time.
    - => shared functionality in other application should respond in time.
  - Network and application availability (reliability of remote connections)
  - Technology choices

Loosely coupled principle
Dependency type examples

1. Runtime environment dependency
   Two components running on the same hardware => 1 component causes a full disk and crashes => The other component stops working too.
2. Technology choice dependency
   A chosen technology for 1 component limits the available technology choices for other components.
3. Time dependency
   The lack of asynchronous interface could make that one component is time dependent to another component. Only during uptime of the systems they can communicate with each other.
4. Data format dependency
   An interface requires a specific data format which differs from the data format in this application.

Software Design Principles with high relevance for the Macro Architecture

High cohesion	Loosely coupled	Separation of concerns	Information hiding principle / Encapsulation
Hierarchical structure	Open closed principle	Keep it simple	Don't repeat yourself

Separation of Concerns principle

Each component is responsible for one function.

Dilemmas:

- Slices or Layers
  Technology layer or Business function is the first criterion to define components ?

- Granularity
  Which seize should application components have at the respective levels?

Separation of Concerns principle ≈ Single Responsibility Principle ≈ Principle of modularity

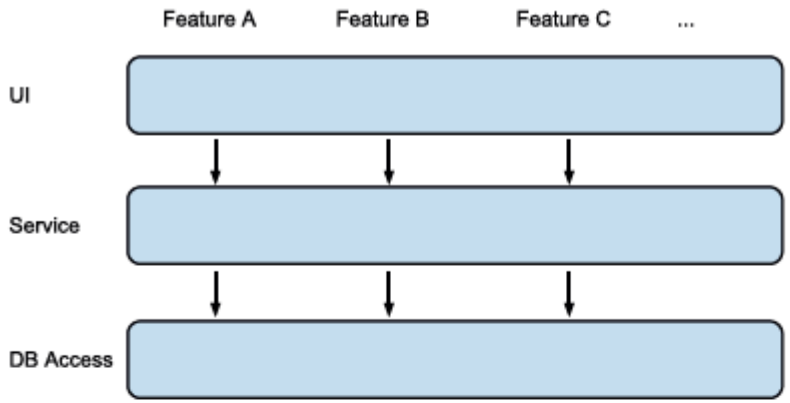


Bild 2.7 Ein System, nach technischen Kriterien in Schichten strukturiert

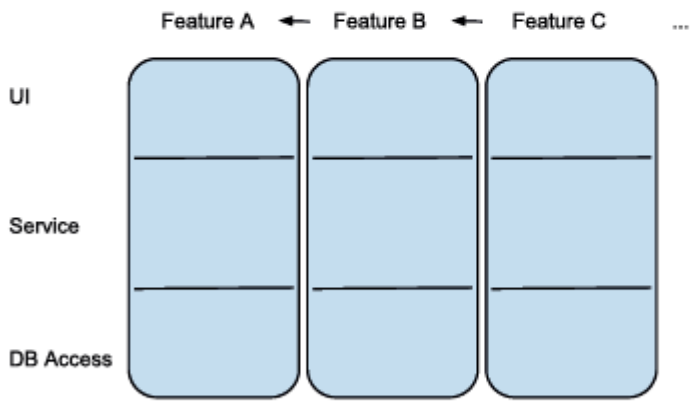


Bild 2.8 Dasselbe System, diesmal in fachlichen Schnitten strukturiert

Wenn Sie ein System in seine Einzelteile zerlegen, tun Sie dies in den oberen Hierarchieebenen am besten nach fachlichen Kriterien. Technische Aspekte sollten dann eher erst in den unteren Ebenen benützt werden, um Strukturen zu bilden.

Separation of Concerns principle
Anti-pattern : Monster class (or God class)

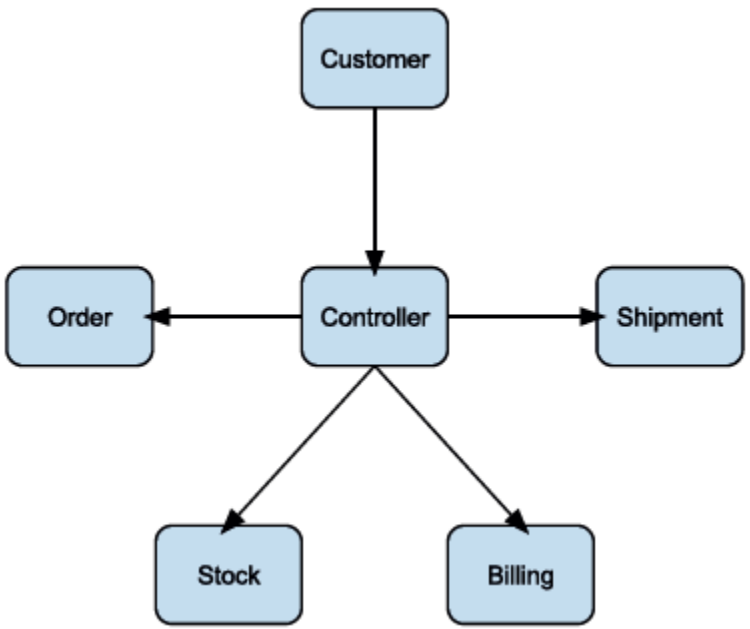
Classroom Assignment:

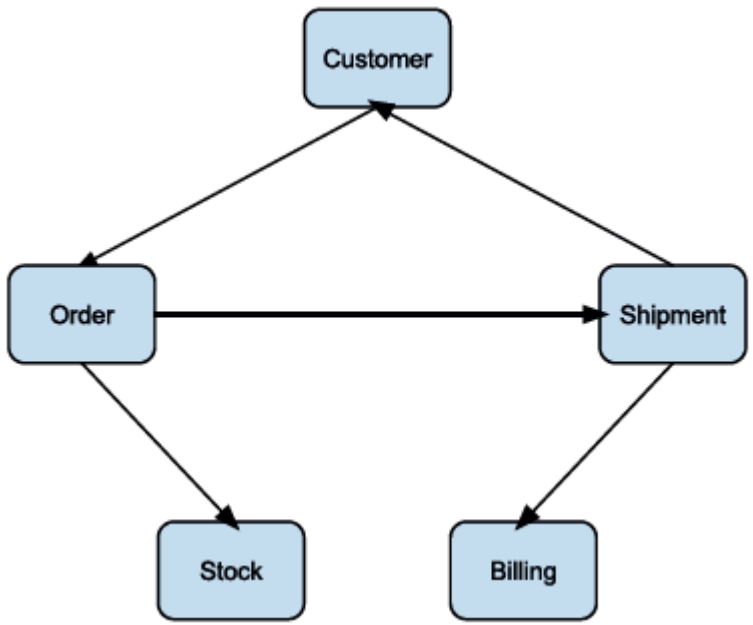
- What is this?
- How is the separation of concerns principle violated?
- And why does this have relevance for the Macro Architecture / Enterprise Application Integration?

"These classes do too much and know too much"

Separation of concerns principle applied to EAI: *"Dumb Pipes and Smart Endpoints"*

Middleware should not hold ~~(all)~~ the business logic


Customer
Order
Controller
Shipment
Stock
Billing


Customer
Order
Shipment
Stock
Billing

Classroom Question:

When you would sin against the principle of Loosely Coupled components, where would the impact be worst?

On the Macro or on the Micro Level?

Software Design Principles with high relevance for the Macro Architecture

High cohesion	★ Loosely coupled	★ Separation of concerns	★ Information hiding principle / Encapsulation
Hierarchical structure	Open closed principle	Keep it simple	Don't repeat yourself

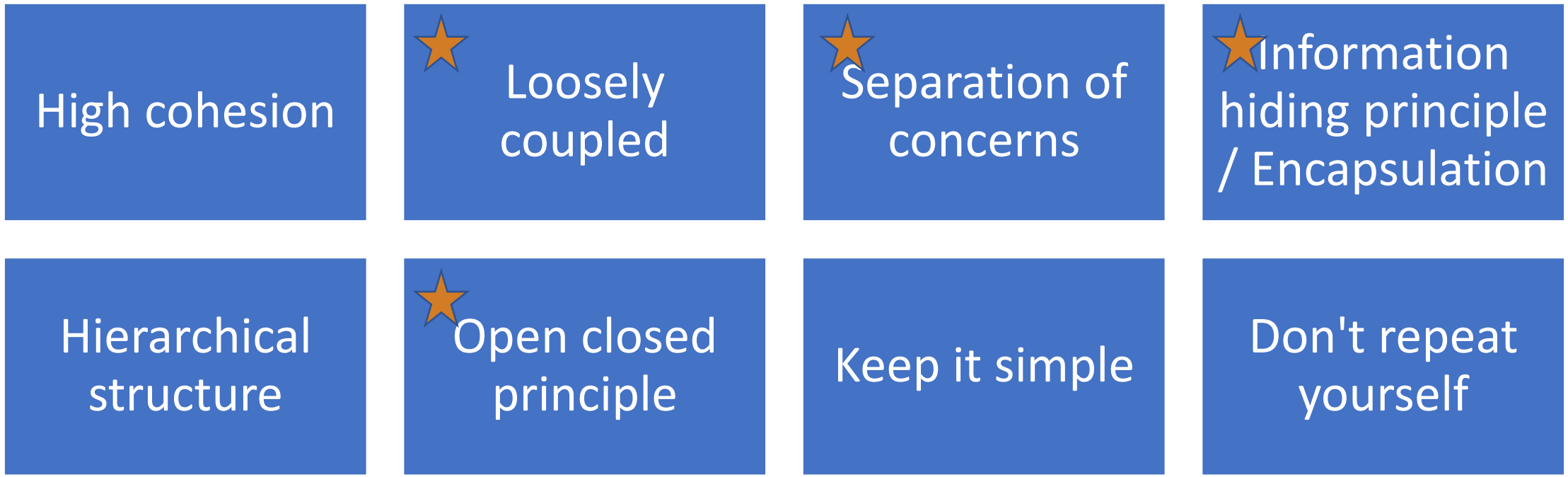
Information Hiding principle & Enterprise Applications

Component inner details are hidden when there is no need for the outside world to know these when using this component.

"If an aspect remains invisible to the outside world, then you have the guarantee that there are no dependencies on this unknown aspect."

=> No dependency to other components / development teams
=> no need discuss / align changes.

Software Design Principles with high relevance for the Macro Architecture



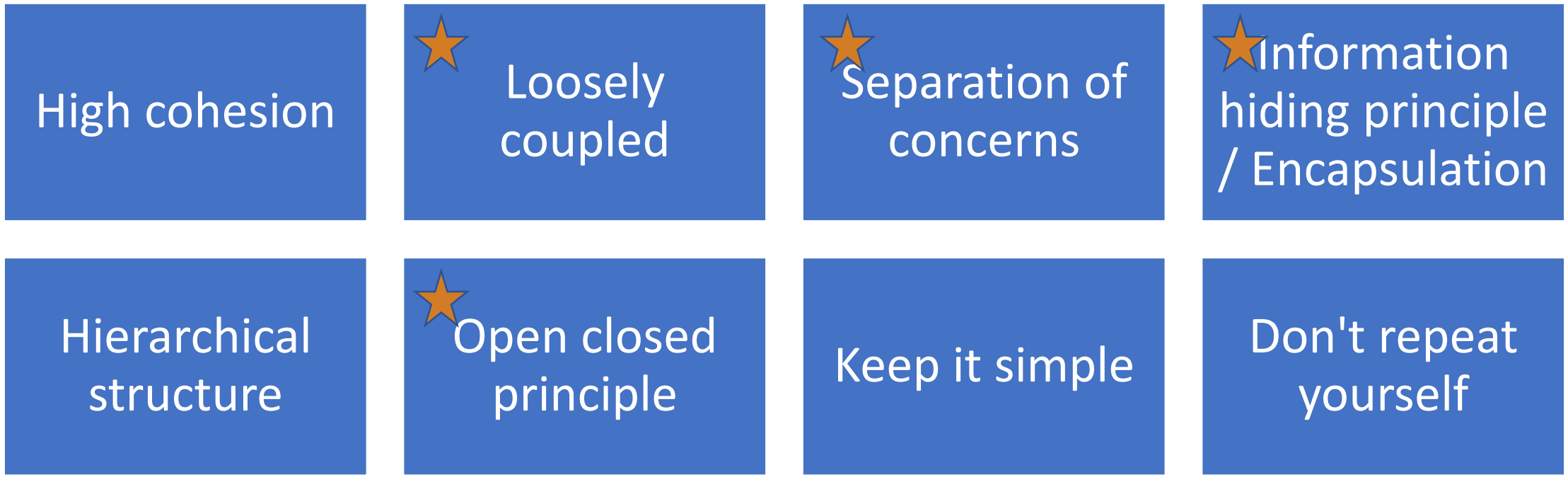
Open Closed principle & Enterprise Applications

Open
It is easy to add a new function. The architecture is open to it.

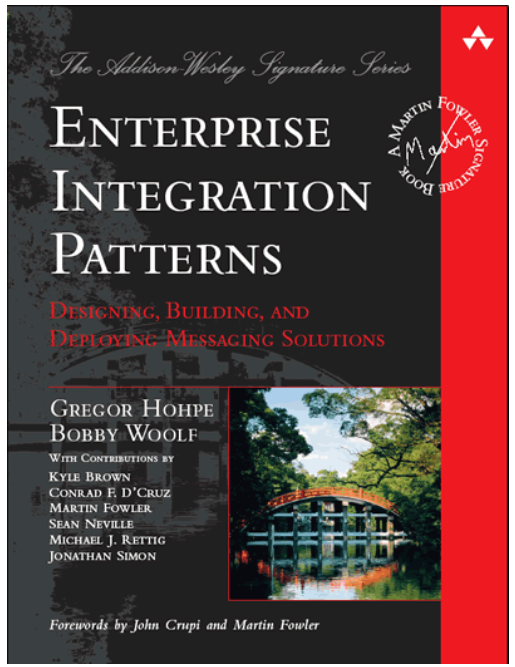
Closed
There is no need to open and change existing components when adding a new function. You can keep these closed.

This principle is very relevant for the Macro Architecture.
When many existing components need to be changed for delivering a new function fast then many teams should plan their work accordingly. This might not be possible at all.

Software Design Principles with high relevance for the Macro Architecture



Mind map for this EAI course



Theory on software design principles & patterns

Knowledge of Enterprise Applications

Theory on EAI messaging patterns

Software Components

Integration

Performance as a whole

+

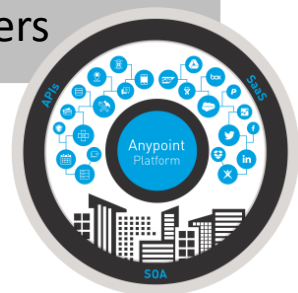
Theory on EAI messaging technologies / standards

Theory on Integration Styles

Knowledge on Middleware products from suppliers

Theory on IT architectures (SOA)

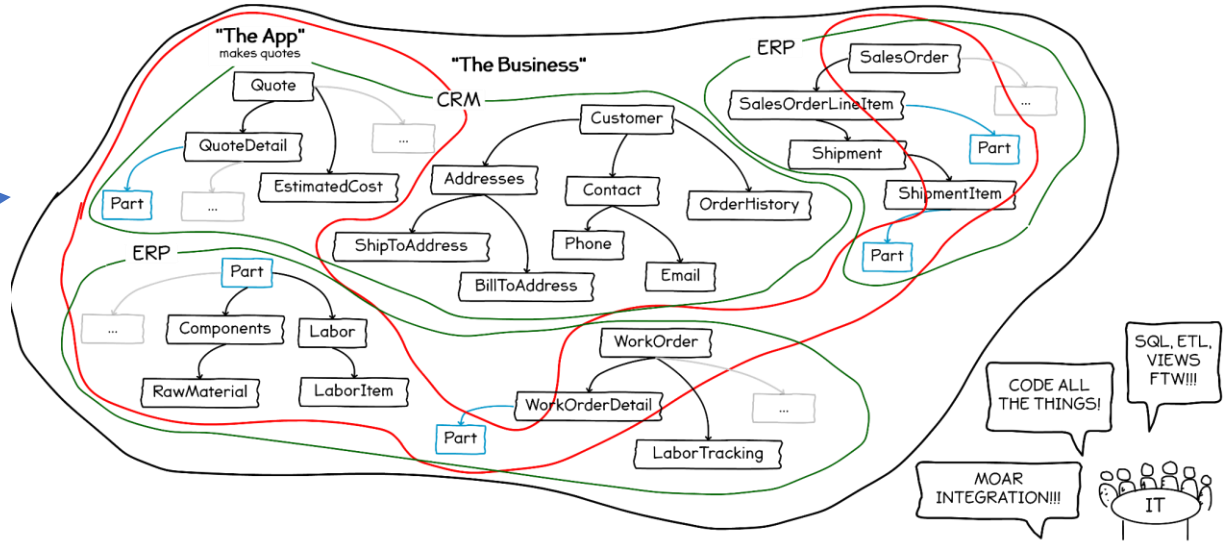
Key Business goal for EAI => Agility which enables short Time to Market

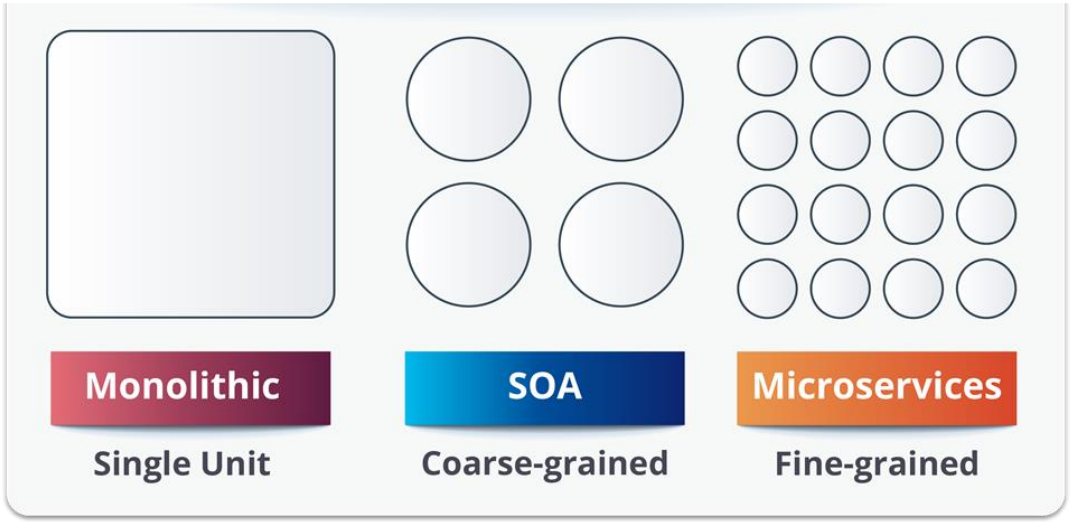


Management summary:
Team dependencies impact business agility

IT Systems landscape architectures, Typology

- Big ball of Mud
- Monolith
- Service Oriented Architecture
- Micro Services
- Hybrid Architecture
- ....





Why does it matter?

A healthy architecture makes it easier to:

- Replace, enhance components fast &
- Fix problems fast

Wrong architectural decisions can have a long lasting negative impact on e.g. costs and agility.

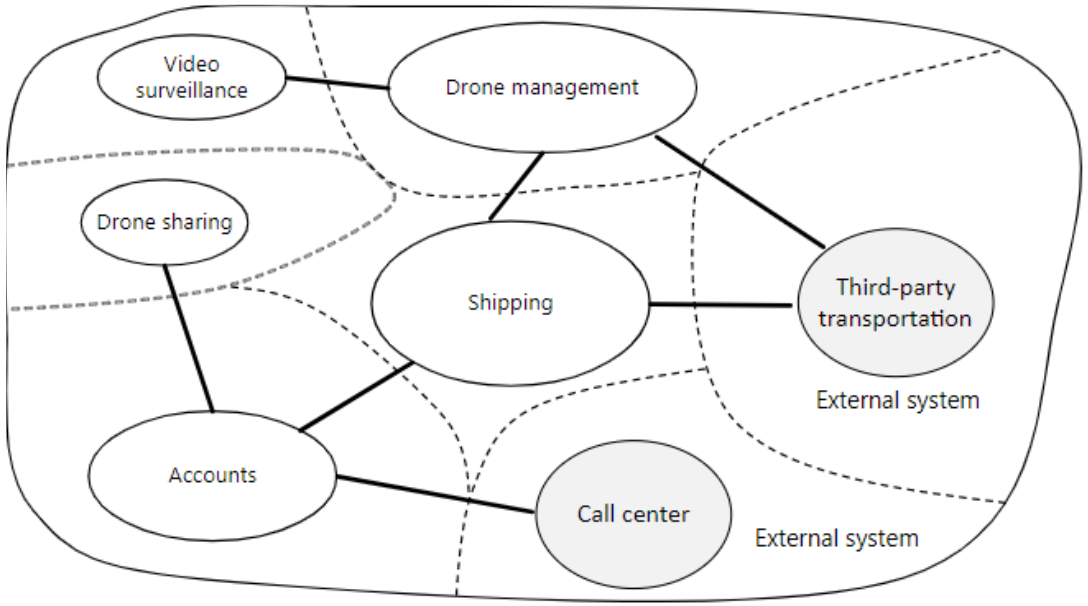
IT Systems landscape architectures, The right cut?

What is the right way to structure the software landscape into components?

Modularization = structure in components

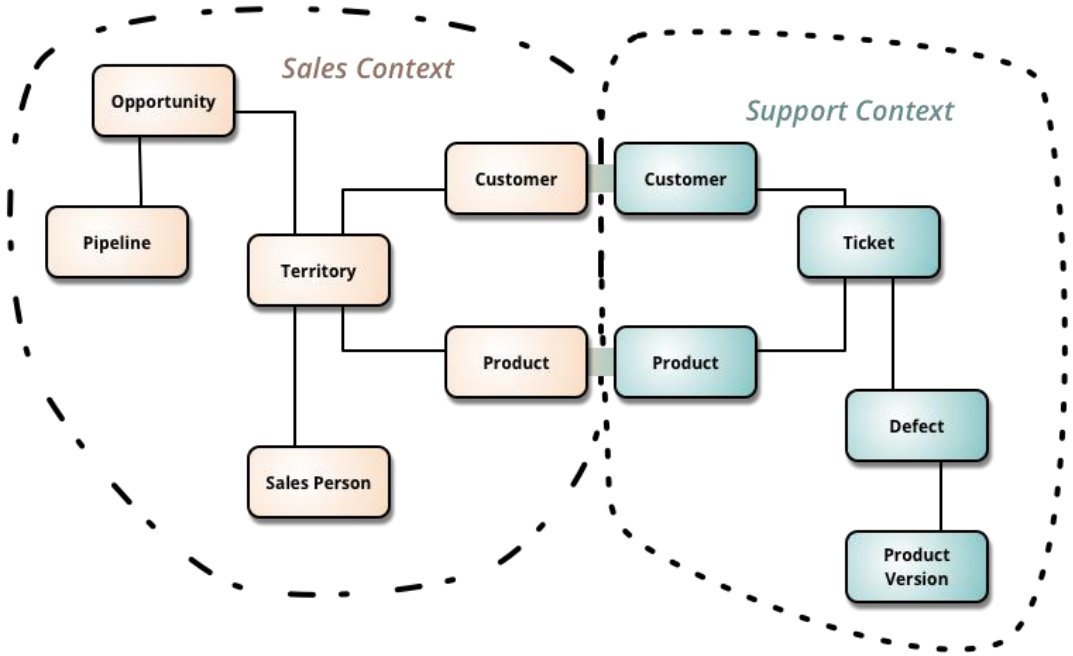
IT Systems landscape architectures, Theoretical Framework(s) to prevent the muddy ball.

- Software design principles
- Domain Driven Design
- Conway's Law



Domain Driven Design

- Ubiquitous Language
  A language structured around the domain model and used by all team members to connect all the activities of the team with the software.[1]
- Bounded Context
  DDD divides up a large system into Bounded Contexts, each of which can have a unified model. Within this context there should be no confusion about the meaning of terms.
- Domain = subject area
- Domain Model



Domain driven design implicitly uses these design principles:

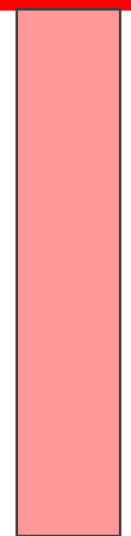
- Separation of Concerns,
- Loosely coupled
- High cohesion

1) Source: https://en.wikipedia.org/wiki/Domain-driven_design

Monolith

- Simplicity of its infrastructure => makes it faster to deploy and scale.
- “only one environment has to be configured to build and deploy the software.”
- “While the complexity may grow over time, **appropriate management** of the code base can help maintain productivity over the lifetime of a monolithic application.”

... can become a complex web of code as the product evolves. Thus, it can be extremely difficult for developers to manage over time.



“It isn’t a bad idea to build a monolithic application, but it is a bad idea to let a monolithic application grow out of control.”

- ”As a result, more time is spent finding the correct line of code, and making sure it doesn’t have side effects.”
- “Monoliths are common because they are simpler to begin building than their alternative, micro services.”

Definition of Microservices architecture

- A service-oriented architecture composed of loosely coupled elements that have bounded contexts.

The concept of *bounded contexts* comes from the book *Domain Driven Design* by Eric Evans.
A microservice with correctly bounded context is self-contained for the purposes of software development.

Example : Mobile App using a Facebook and Google Maps micro service.

When you update the app you do not have to talk to the development teams of Facebook and Google

Some consider Microservices as *"SOA done right"*

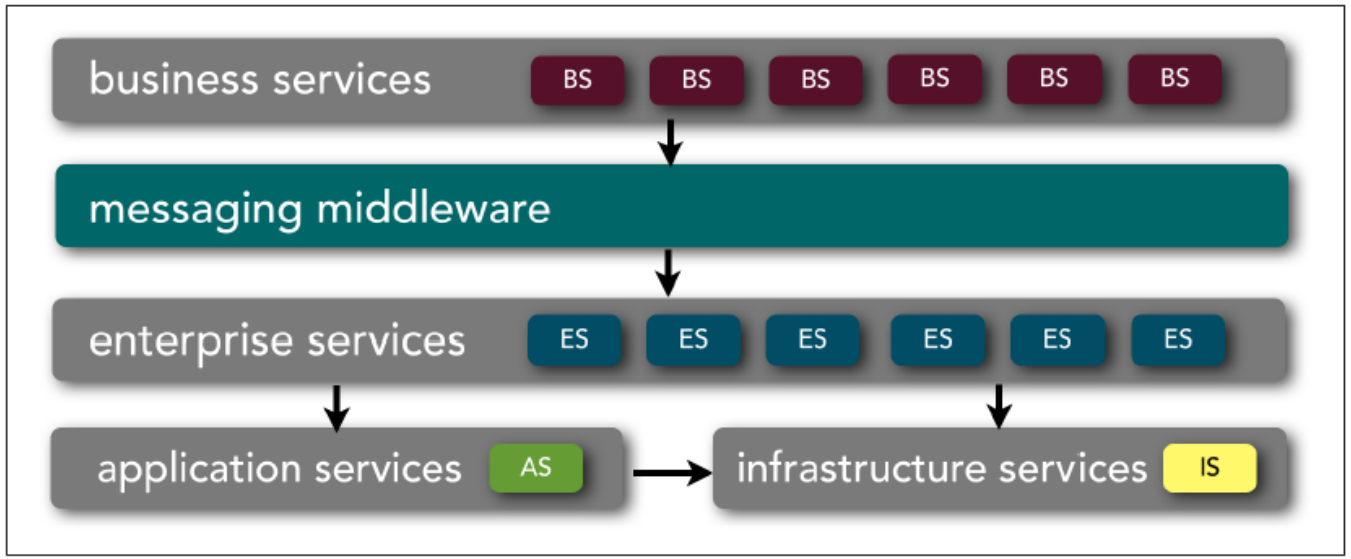


Figure 2-2. SOA taxonomy

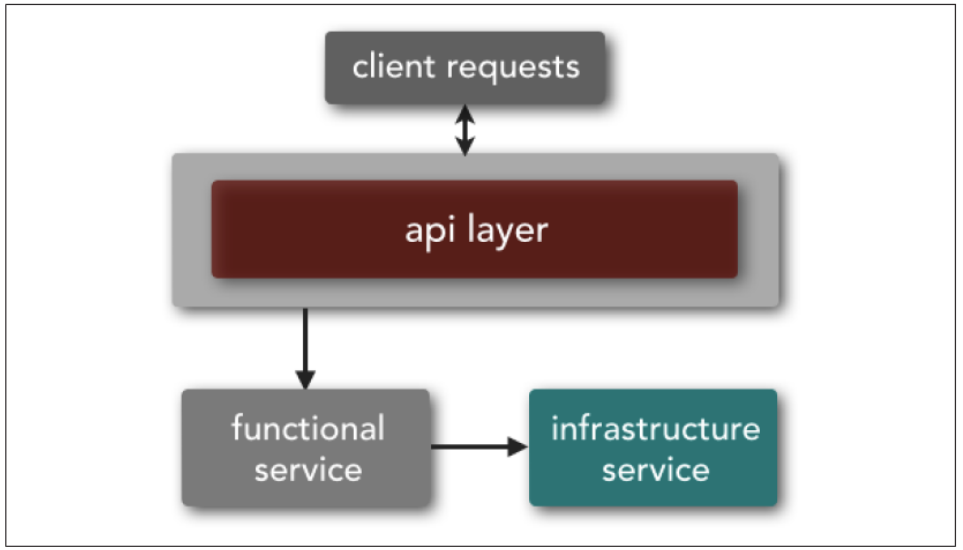


Figure 2-1. Microservice service taxonomy

Now think of teams!

Now think of teams!

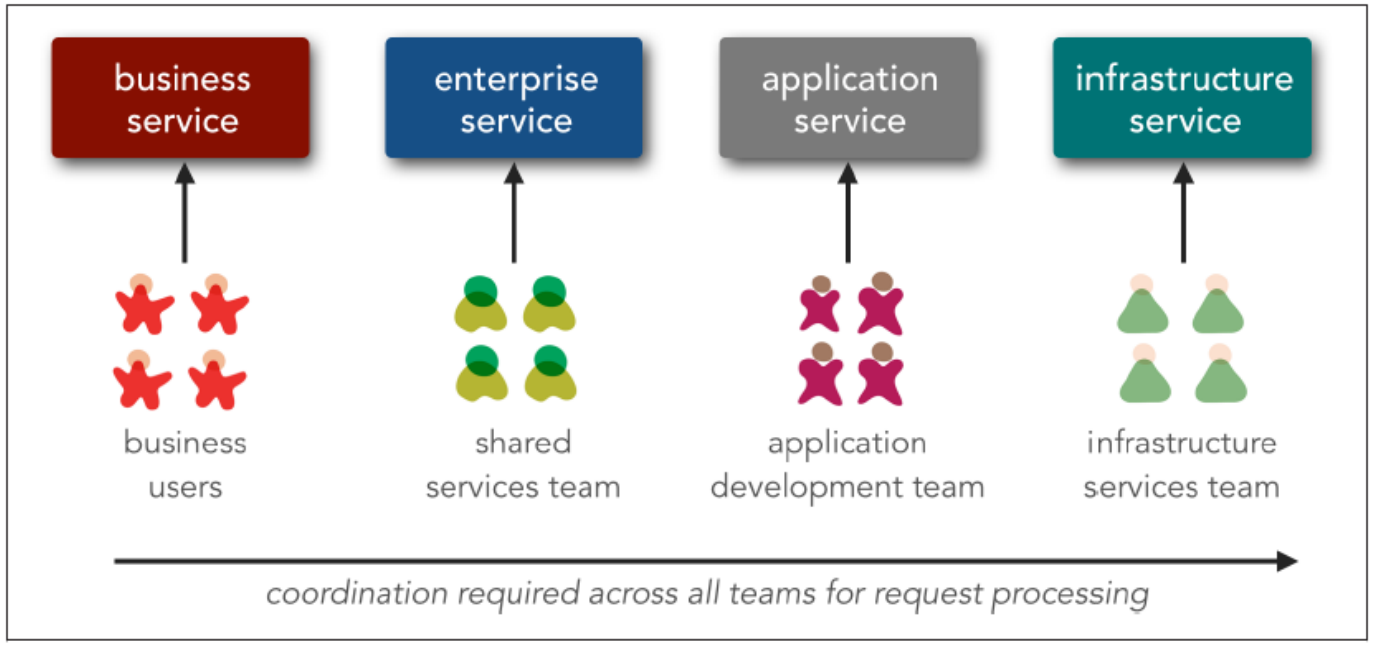


Figure 2-4. SOA service ownership model

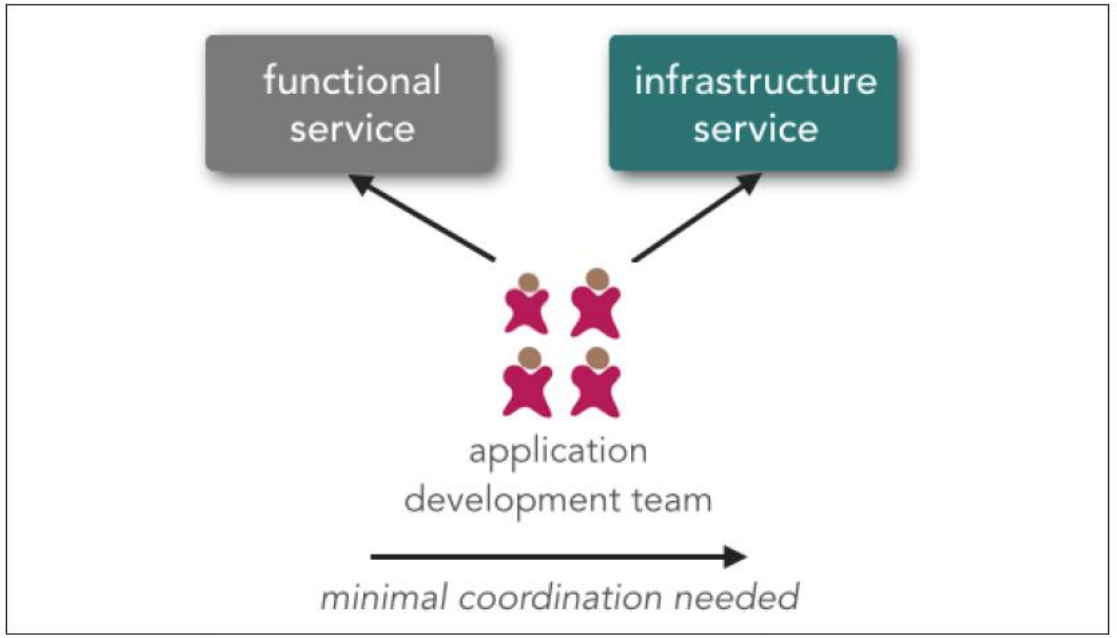
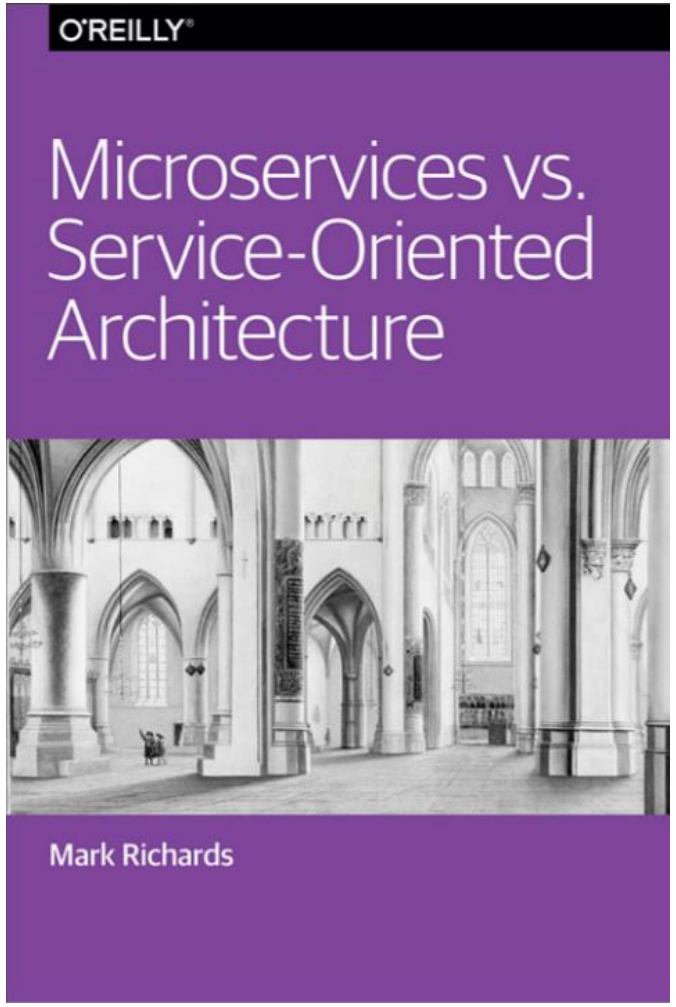


Figure 2-3. Microservices service ownership model

Further reading .....



- O'Reilly !!!!
  http://www.oreilly.com/programming/free/files/microservices-vs-service-oriented-architecture.pdf
- Amazon website !
  https://aws.amazon.com/microservices/
- Martin Fowler
  https://martinfowler.com/articles/microservices.html

Watch this!

Microservices architecture at Netflix

- https://youtu.be/CriDUYtfrjs?t=153

Start at 2:33

And please note Netflix Open Source Stack (min 40:48)