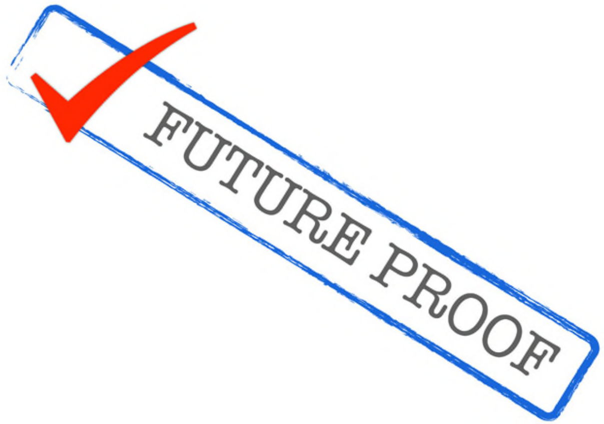


https://static1.squarespace.com

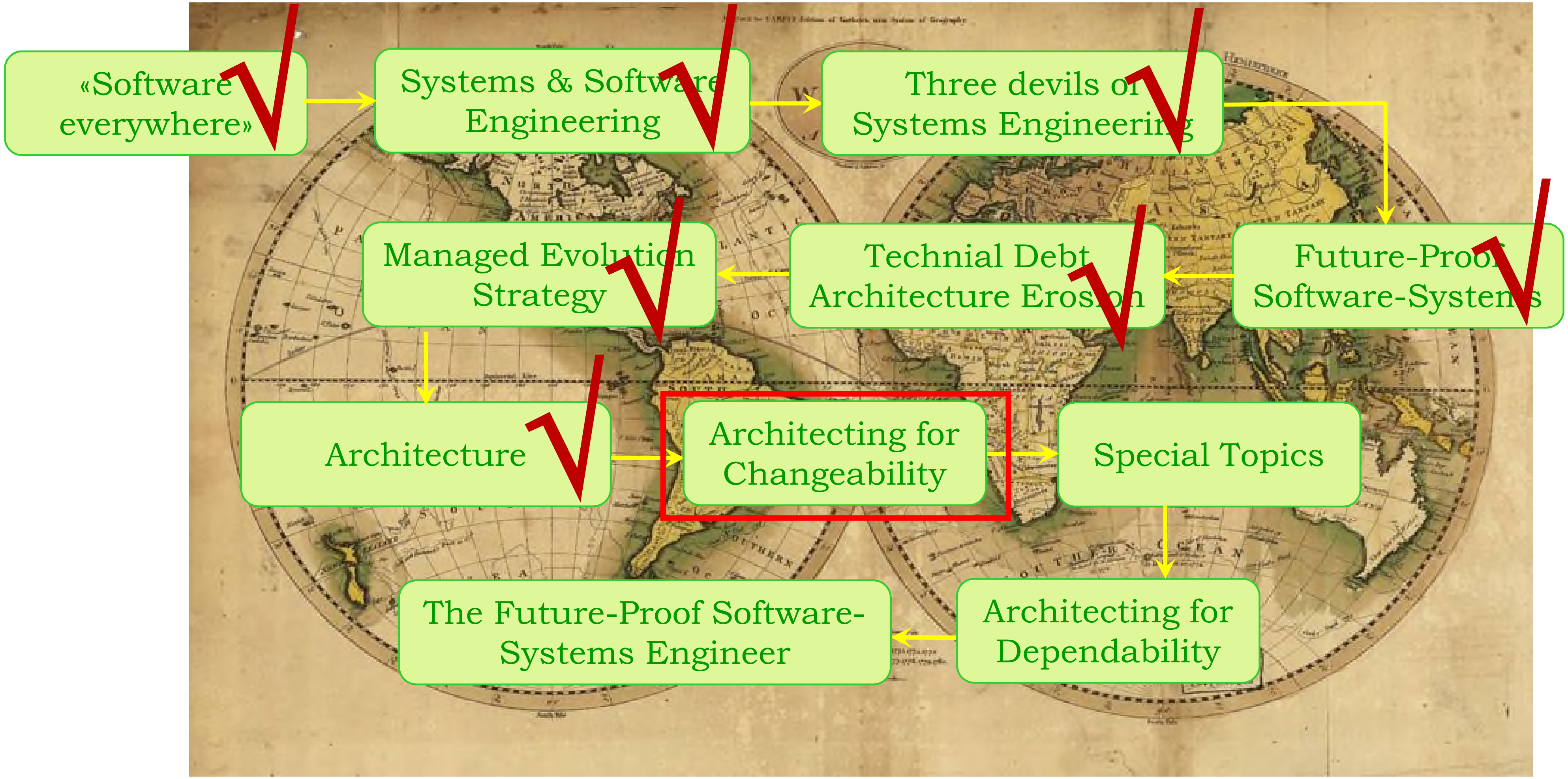
Future-Proof Software-Systems (FPSS)

Part 3A: Architecting for Changeability

Lecture WS 2017/18: Prof. Dr. Frank J. Furrer

Version 1.3

Our journey:


«Software everywhere»
Systems & Software Engineering
Three devils of Systems Engineering
Future-Proof Software-Systems
Technial Debt Architecture Erosion
Managed Evolution Strategy
Architecture
Architecting for Changeability
Special Topics
Architecting for Dependability
The Future-Proof Software-Systems Engineer

Changeability: Repetition

Changeability is the capability of an **organization** to develop software-systems:

- With high-quality functionality
- With a good cost and time-to-market performance

The most important key factor for changeability is the **structure** of the the software

Architecture!

Changeability

Metric

high

low

Development Cost per Unit of Functionality

4.2 k€/UCP

0.8 days/UCP

Company A

Time-to-Market per Unit of Functionality

10.0 k€/UCP

4.0 days/UCP

Company B

50.0 k€/UCP

10.0 days/UCP

Company C

Project
250 UCP

Company A
Project Cost:
1’050 k€
Time to Market:
200 days

Company B
Project Cost:
2’500 k€
Time to Market:
1’000 days

Company C
Project Cost:
12’500 k€
Time to Market:
2’500 days

http://www.clipartsheep.com

http://www.clipartshut.com

http://www.clipartpanda.com

http://www.0lll.com/architecture-exhibitions/?gal=24

http://www.asisbiz.com/index.html

Fact 1:

Good **architecture** results in good **changeability**

Part 3

Fact 2:

Good **architecture** is governed by proven **architecture principles**

http://4.bp.blogspot.com

https://upload.wikimedia.org

Fact 1:
Good **architecture** results in good **dependability**

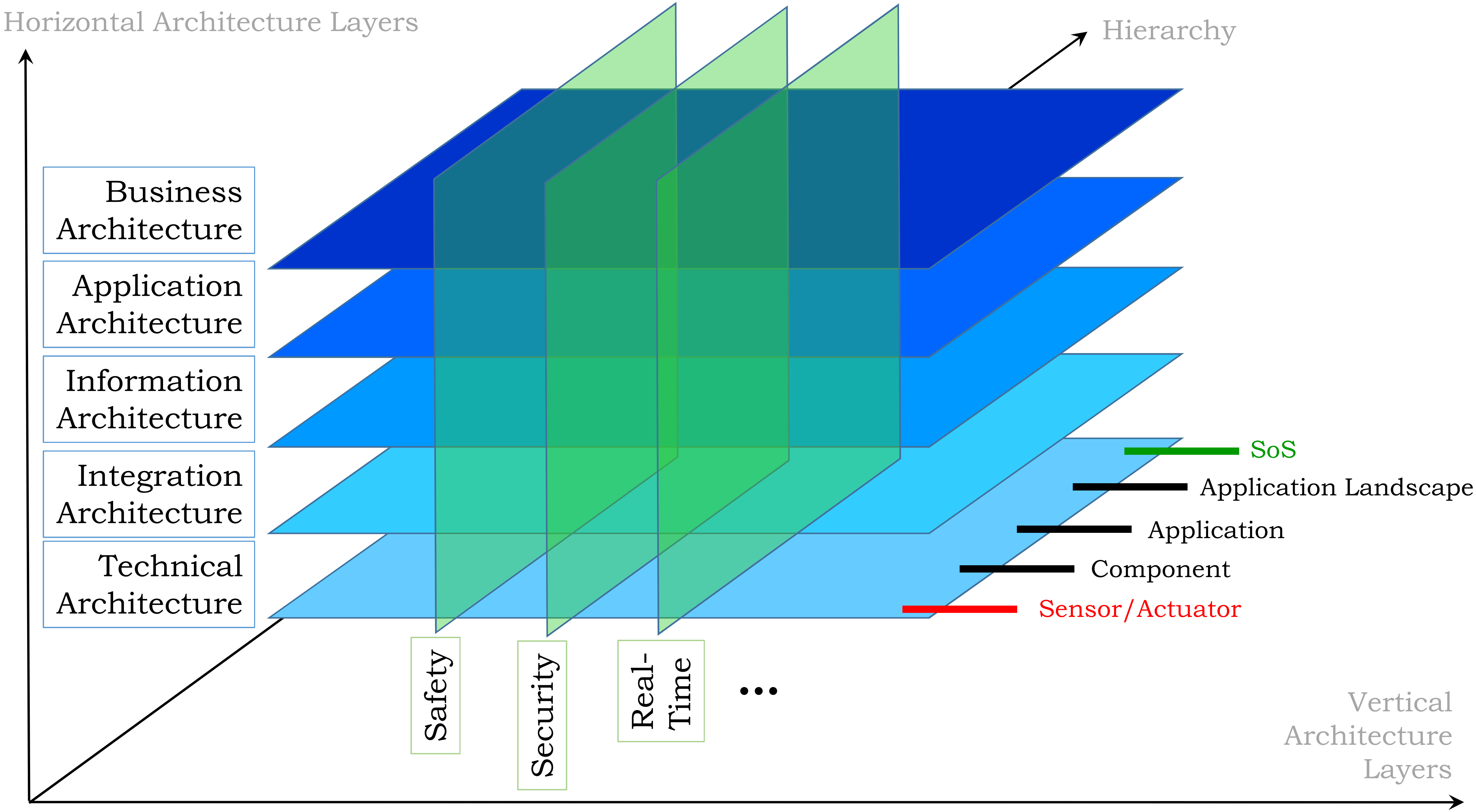
Fact 2:
Good **architecture** is governed by proven **architecture principles**

Part 4

https://metrouk2.files.wordpress.com

[Information Systems] Architecture Principles

for Changeability	for Dependability	for other Attributes
12 architecture principles [complete set]	Resilience: 9 principles [complete set] Dependability: Examples	Examples


Horizontal Architecture Layers
Hierarchy
Business Architecture
Application Architecture
Information Architecture
Integration Architecture
Technical Architecture
SoS
Application Landscape
Application
Component
Sensor/Actuator
Safety
Security
Real-Time
...
Vertical Architecture Layers

Horizontal Architecture Layers

Business
Architecture
(Business Processes)

Applications
Architecture
(Functionality)

Information (Data)
Architecture
(Information & Data)

Integration
Architecture
(Cooperation
Mechanisms)

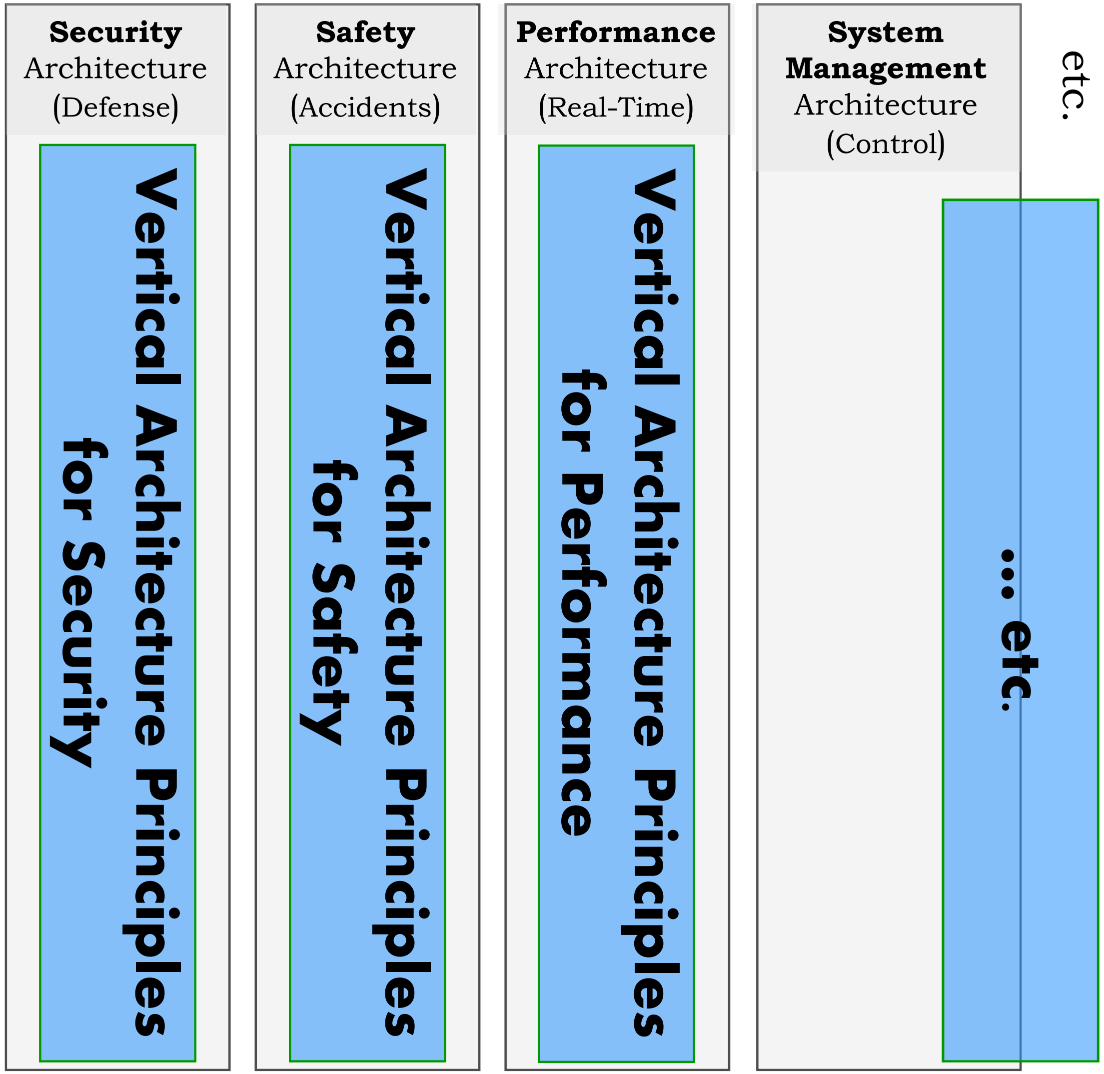
Technical
Architecture
(Technical
Infrastructure)

Horizontal Architecture Principles
(for Changeability)

Vertical Architecture Layers

Vertical Architecture Principles for **Dependability**

… and other quality attributes



Architecture Principles for Changeability

Objective: Provide a set of *Architecture Principles* which lead to high **changeability**

Engineering Discipline: Principle-based Architecting

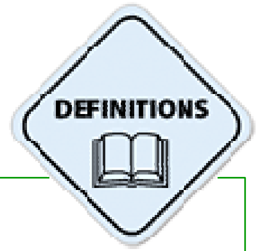
https://disappearinginplainsight.com

… we need to:

- understand,
- consistently apply,
- and strongly enforce

the architecture principles

You will learn: Proven Architecture Principles for the Construction of Future-Proof Software-Systems



Architecture Principles

Fundamental insights
– formulated as *enforcable* **rules** –
how future-proof software-systems should be built

http://www.telco2.net/blog/2007/04/telco_20_event_digital_worker.html

http://www.ledvision.com

Horizontal Architecture Layer Principles:

- A1: Architecture Layer Isolation
- A2: Partitioning, Encapsulation and Coupling
- A3: Conceptual Integrity
- A4: Redundancy
- A5: Interoperability
- A6: Common Functions
- A7: Reference Architectures, Frameworks and Patterns
- A8: Reuse and Parametrization
- A9: Industry Standards
- A10: Information Architecture
- A11: Formal Modeling
- A12: Complexity and Simplification

https://us.123rf.com

Fundamental Principles:

12

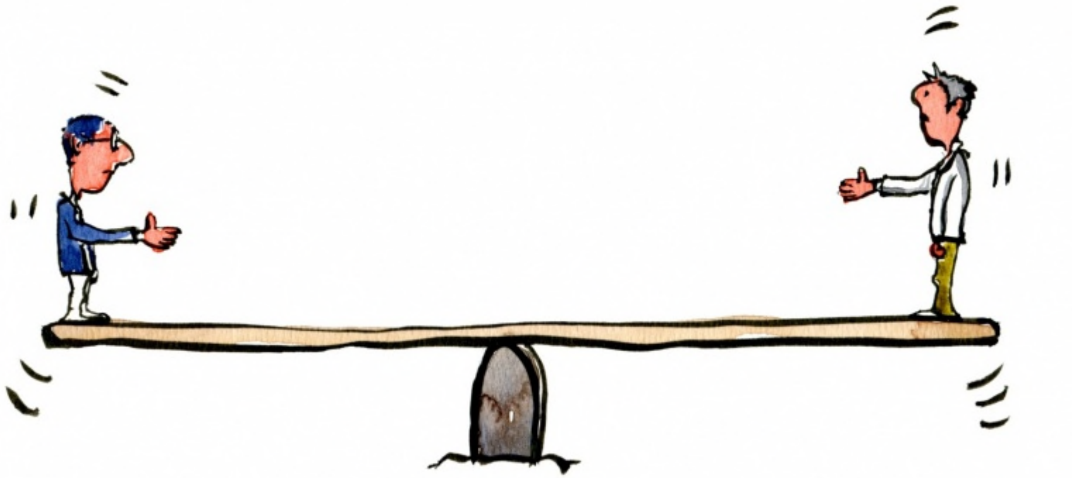
for Changeability

(presented in this lecture)

The architecture principles are strongly worded rules, often using «**never**» or «**always**»

Are they always - without exceptions - to be followed?

… however:



http://art.fritsahlefeldt.com

Sometimes compromises are necessary
(more about later)

https://biblepaedia.files.wordpress.com

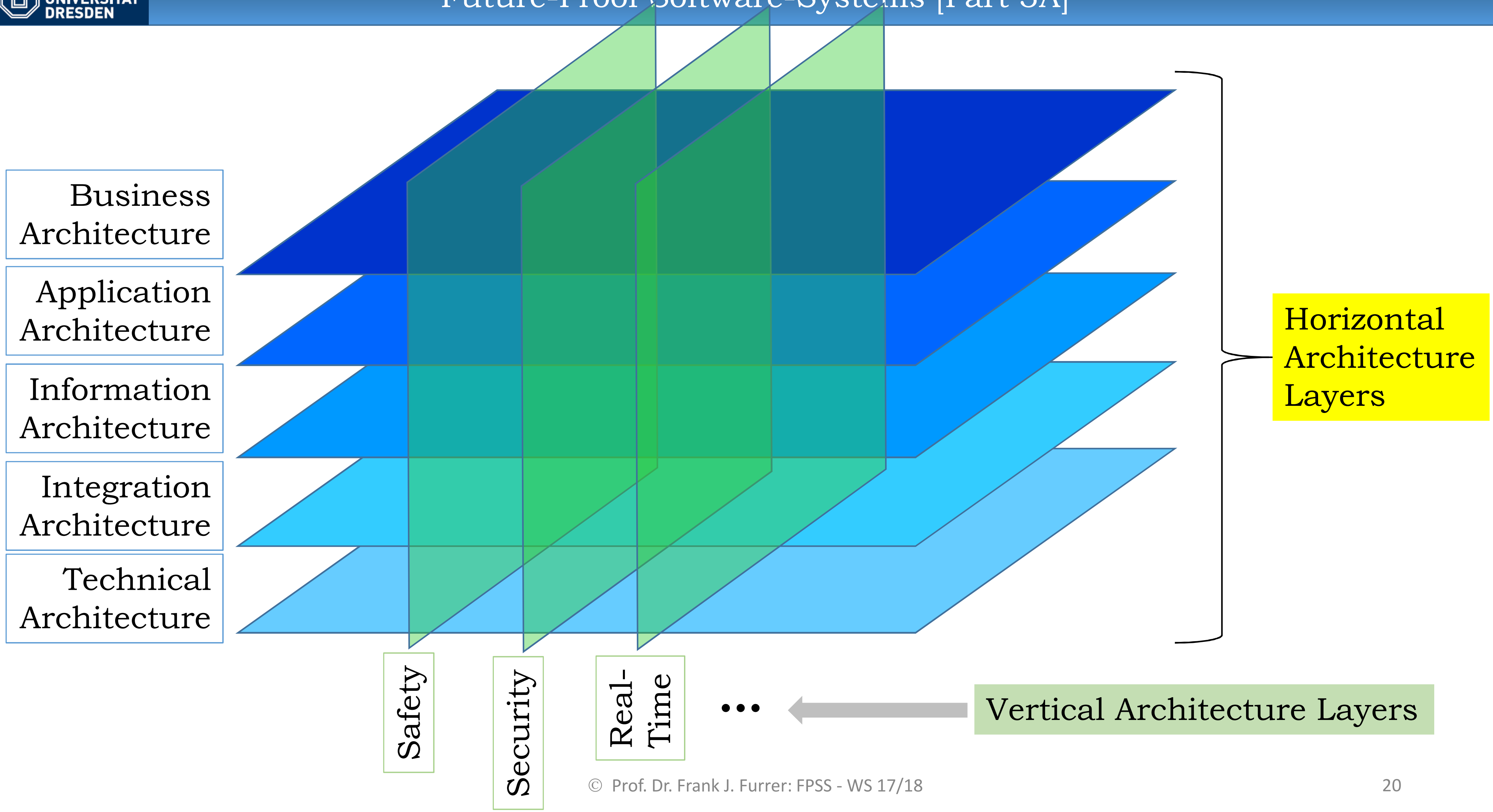
A1

Architecture Principle A1:

Architecture Layer Isolation

Horizontal Architecture Layer Principles:

- A1: Architecture Layer Isolation
- A2: Partitioning, Encapsulation and Coupling
- A3: Conceptual Integrity
- A4: Redundancy
- A5: Interoperability
- A6: Common Functions
- A7: Reference Architectures, Frameworks and Patterns
- A8: Reuse and Parametrization
- A9: Industry Standards
- A10: Information Architecture
- A11: Formal Modeling
- A12: Complexity and Simplification


Business Architecture
Application Architecture
Information Architecture
Integration Architecture
Technical Architecture
Horizontal Architecture Layers
Safety
Security
Real-Time
…
Vertical Architecture Layers

Key Idea:

Business Architecture Layer
(Business Processes)

Isolation

Applications Architecture Layer
(Functionality)

Isolation

Information (Data) Architecture Layer
(Information & Data)

Isolation

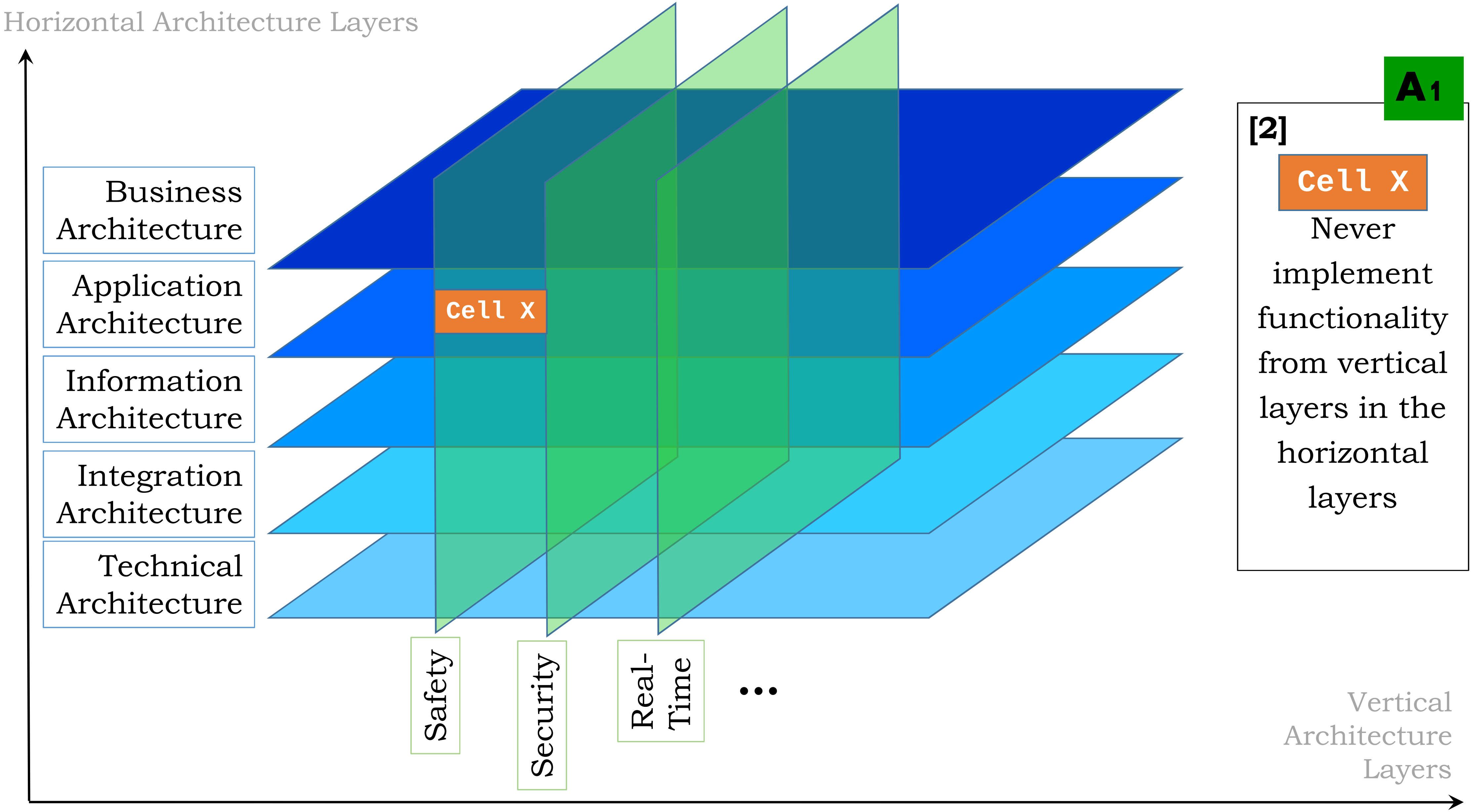
Integration Architecture Layer
(Cooperation Mechanisms)

Isolation

Technical Architecture Layer
(Technical Infrastructure)

A1

[1]
Always use standardized, technology-independent, and product-independent mechanisms for transfer of data and control between layers


Horizontal Architecture Layers
Business Architecture
Application Architecture
Information Architecture
Integration Architecture
Technical Architecture
Cell X
Safety
Security
Real-Time
...
A1
[2]
Cell X
Never implement functionality from vertical layers in the horizontal layers
Vertical Architecture Layers

Breaking Layers

Business Architecture **Layer**
(Business Processes)

Isolation

Applications Architecture **Layer**
(Functionality)

Isolation

Information (Data) Architecture **Layer**
(Information & Data)

Isolation

Integration Architecture **Layer**
(Cooperation Mechanisms)

Isolation

Technical Architecture **Layer**
(Technical Infrastructure)

Direct access – **bypassing** the standardized, technology-independent mechanisms

Result:

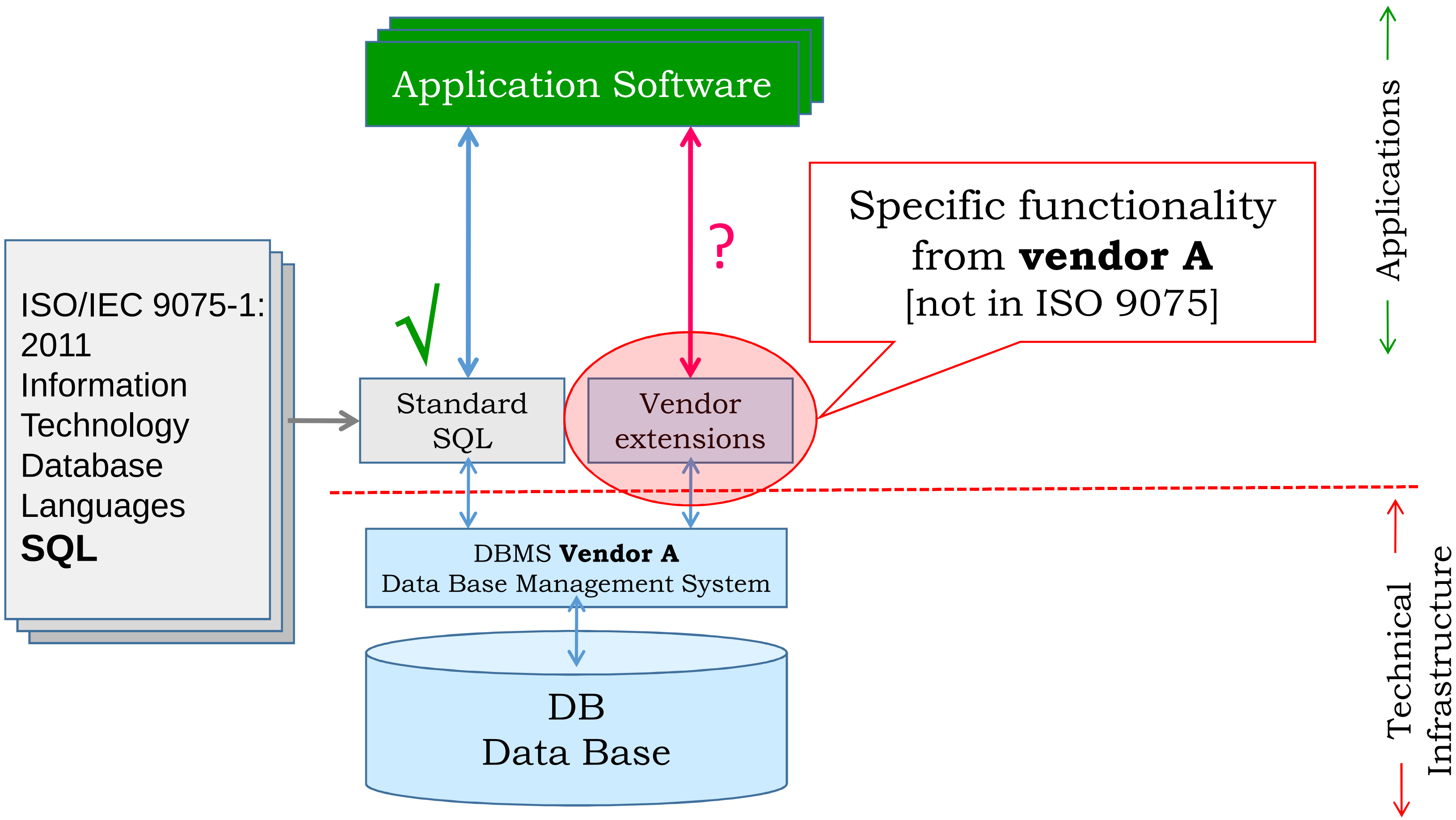
- Technology dependence
- Vendor lock-in
- No standards-compliance

http://media.istockphoto.com

Example: Misuse of SQL (1/2)

[SQL = Structured Query Language]

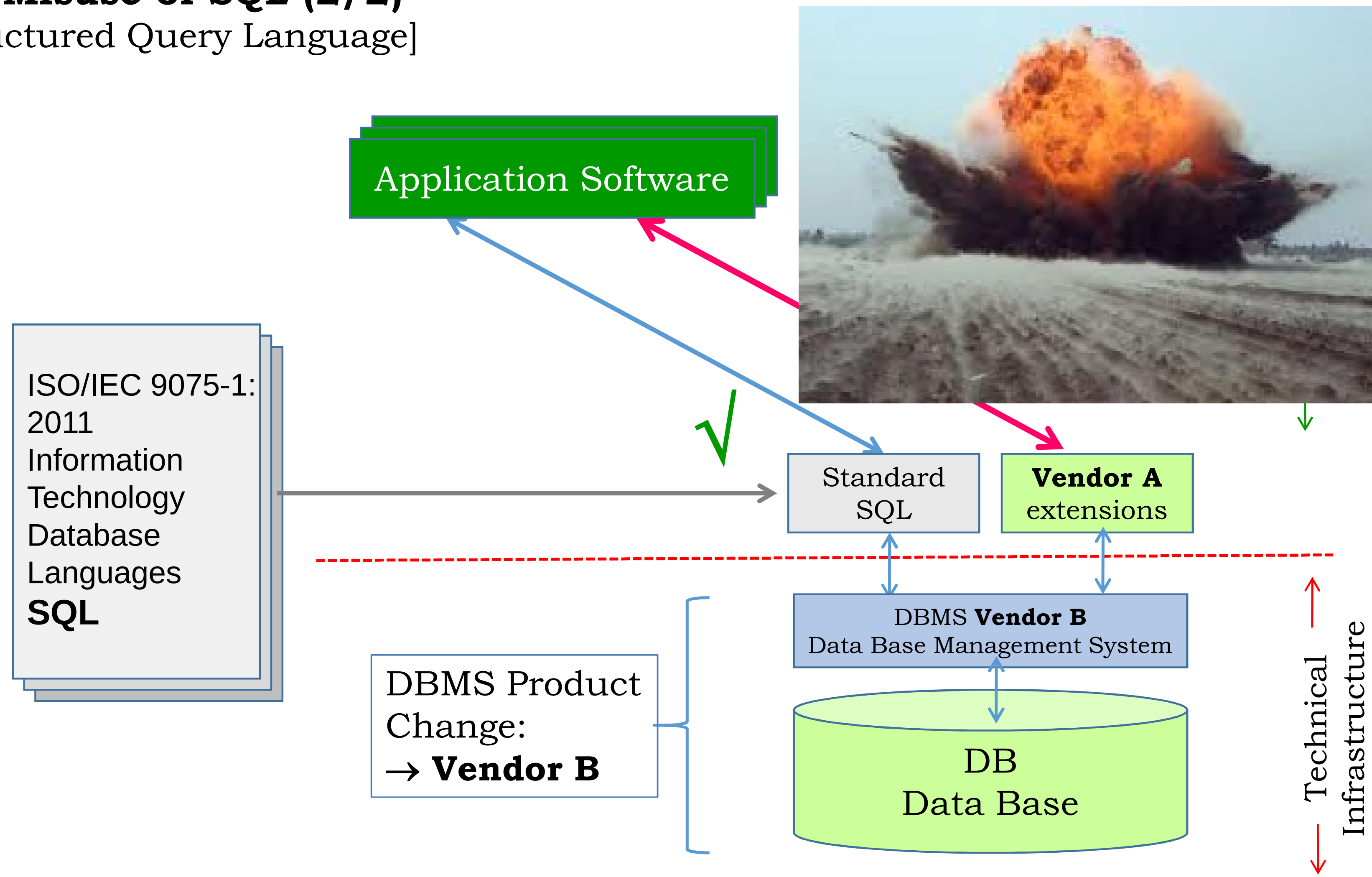
Breaking Layers



Example: Misuse of SQL (2/2)

[SQL = Structured Query Language]

www.beta.jootix.com



Business Architecture **Layer**
(Business Processes)

Isolation

Applications Architecture **Layer**
(Functionality)

Isolation

Information (Data) Architecture **Layer**
(Information & Data)

Isolation

Integration Architecture **Layer**
(Cooperation Mechanisms)

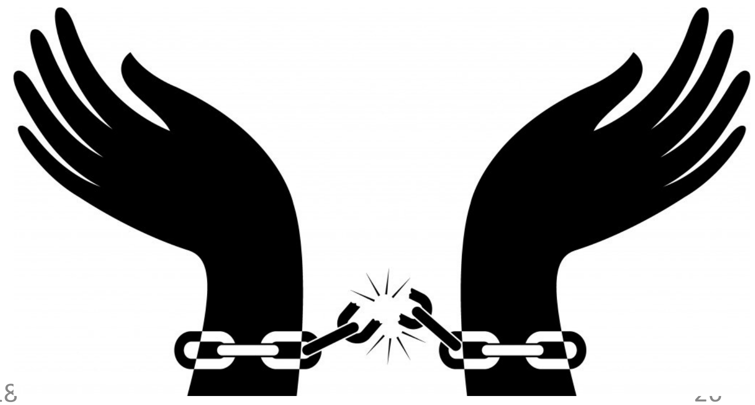
Isolation

Technical Architecture **Layer**
(Technical Infrastructure)

NOT Breaking Layers

Industry-standard, technology-independent mechanism

Result:
- Technology **in**dependence
- Vendor **in**depence
- Full standards-compliance

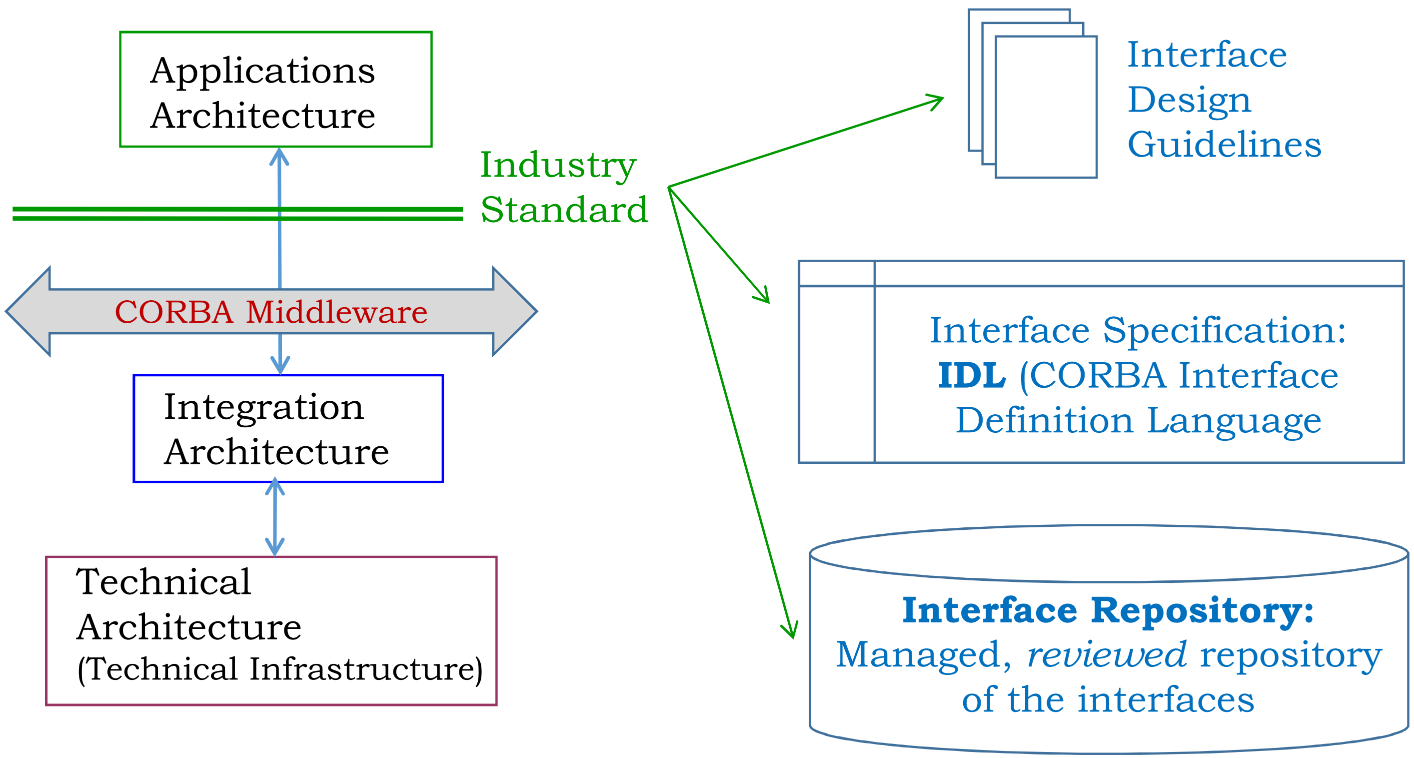


https://i1.wp.com

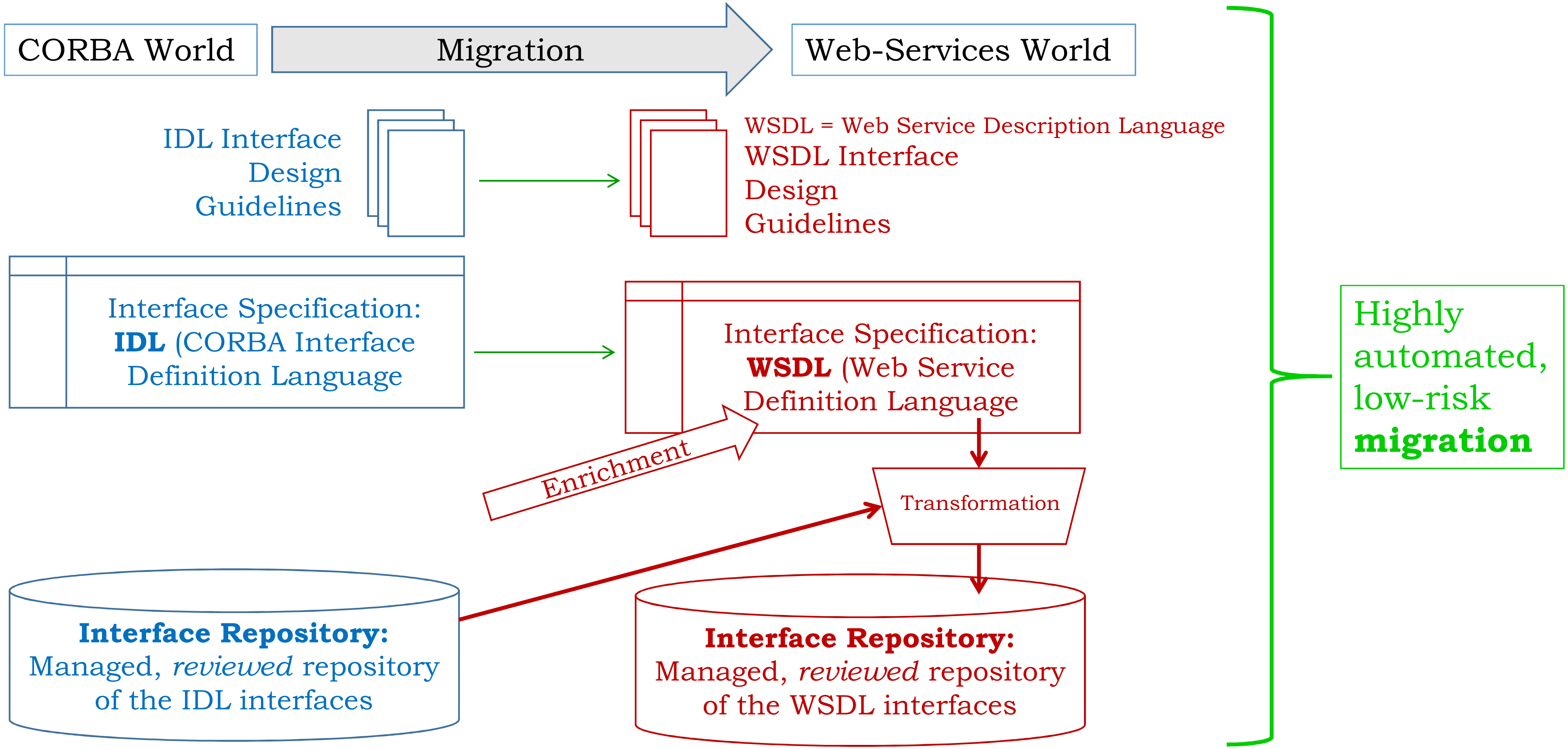
Example: CORBA-services to Web-services migration (1/2)

[CORBA = Common Object Request Broker Architecture]

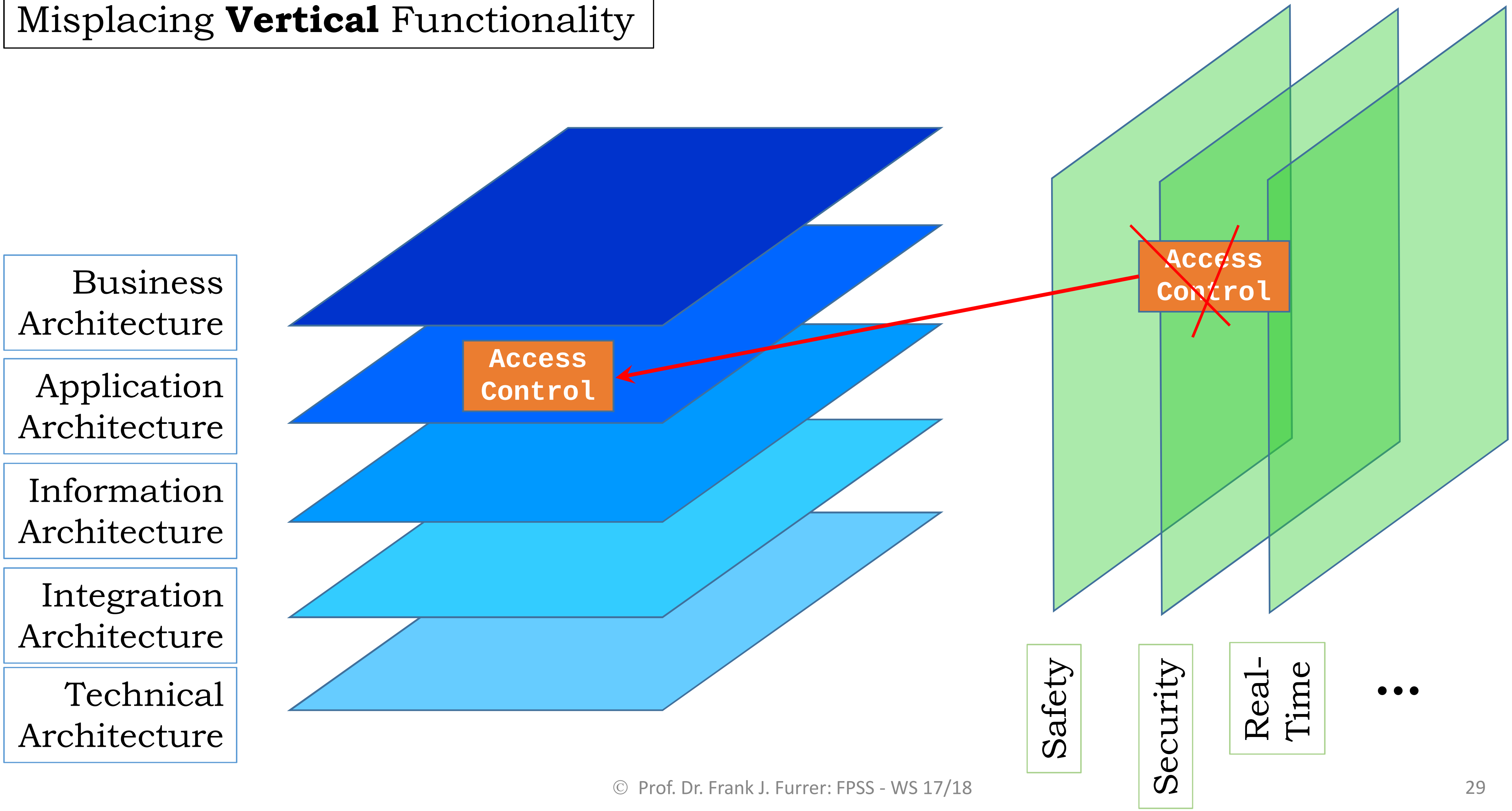
NOT Breaking Layers



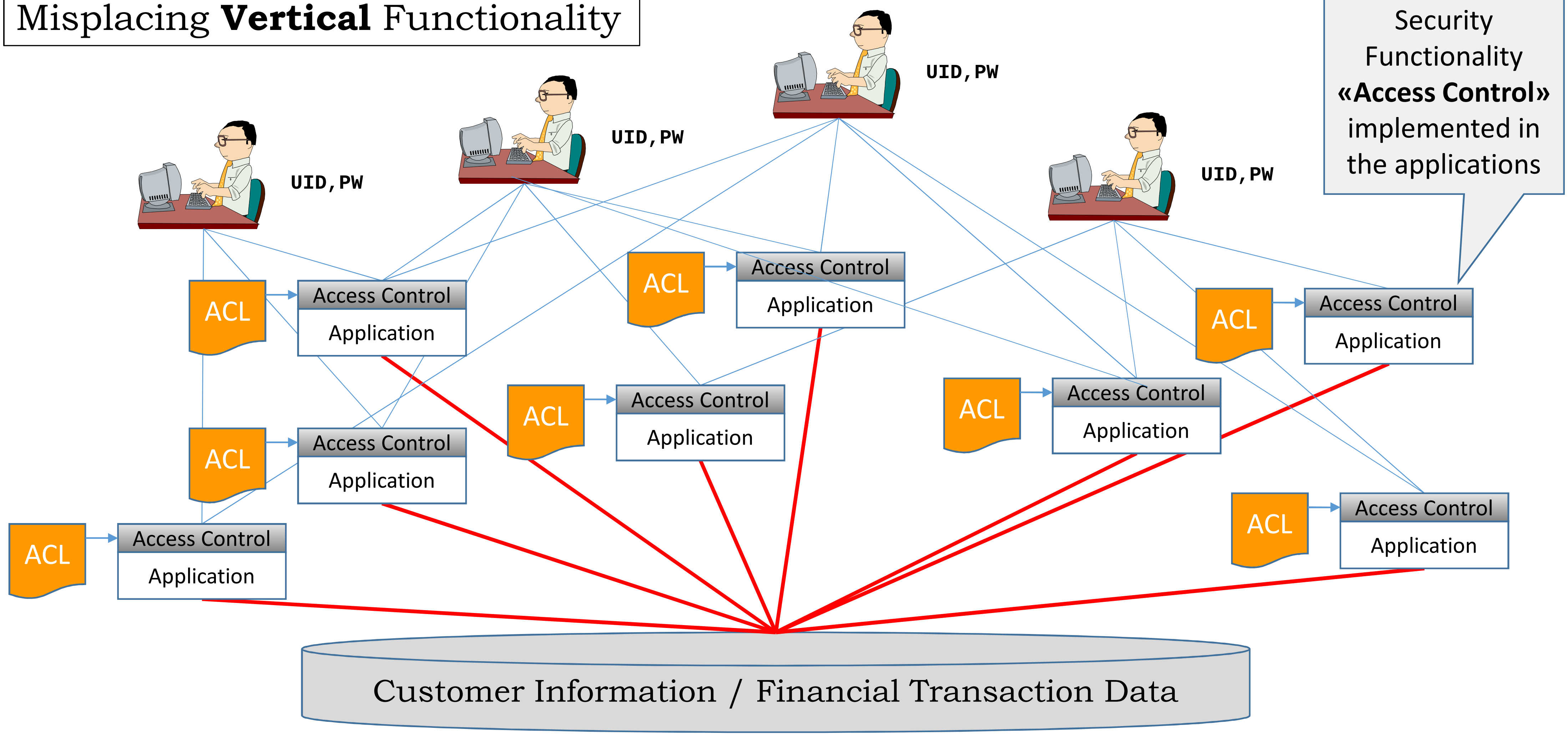
Example: CORBA-services to Web-services migration (2/2)



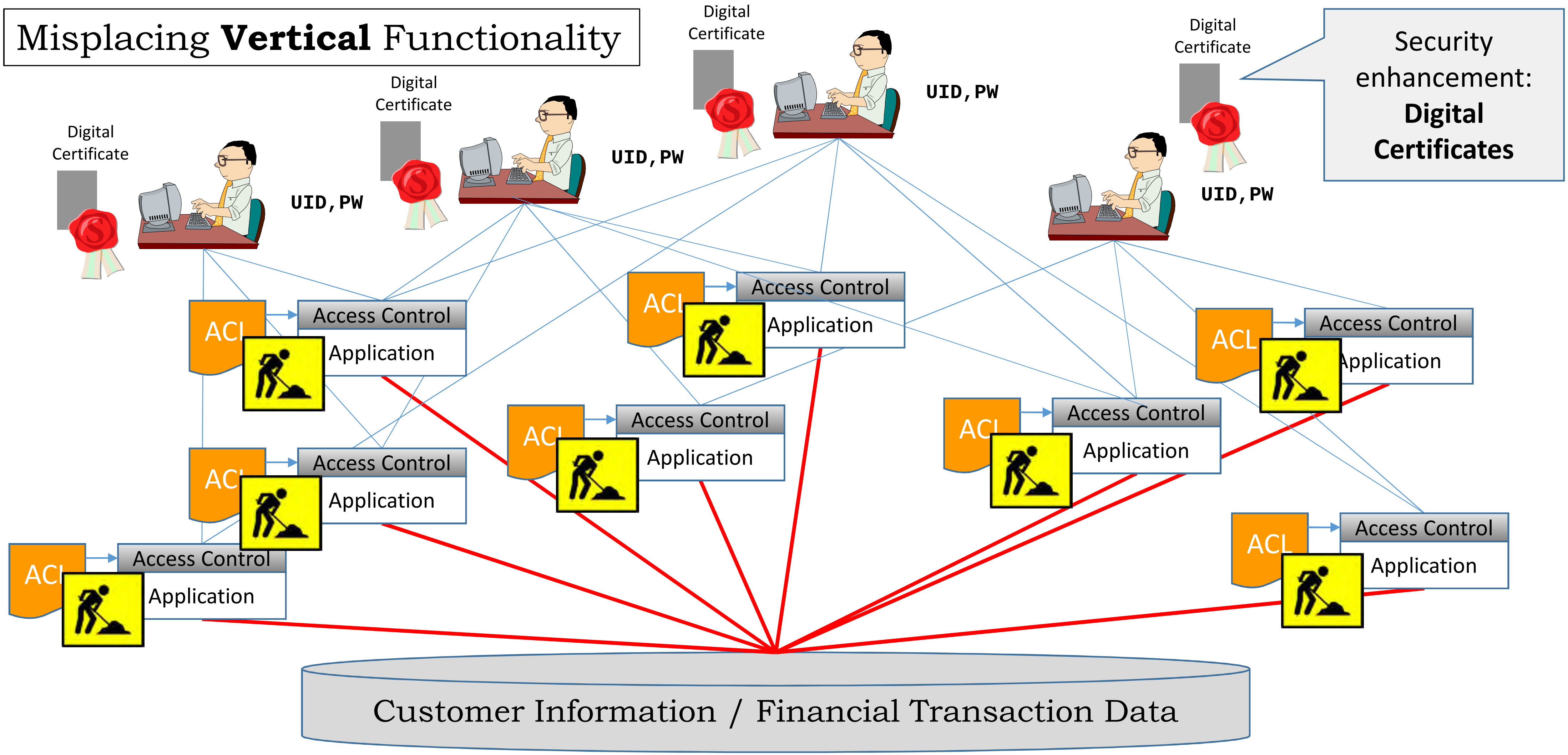
Misplacing **Vertical** Functionality



Misplacing **Vertical** Functionality

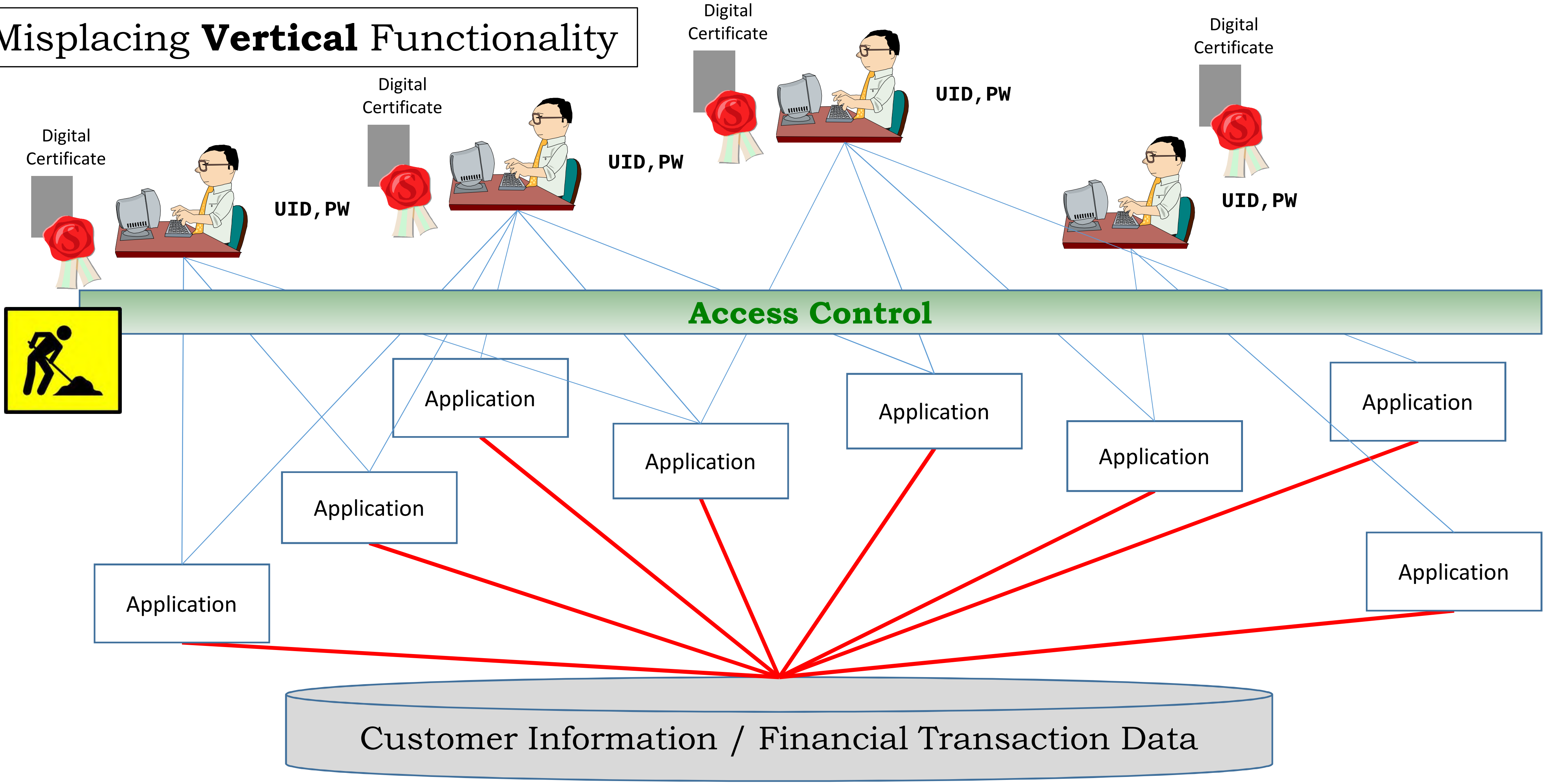


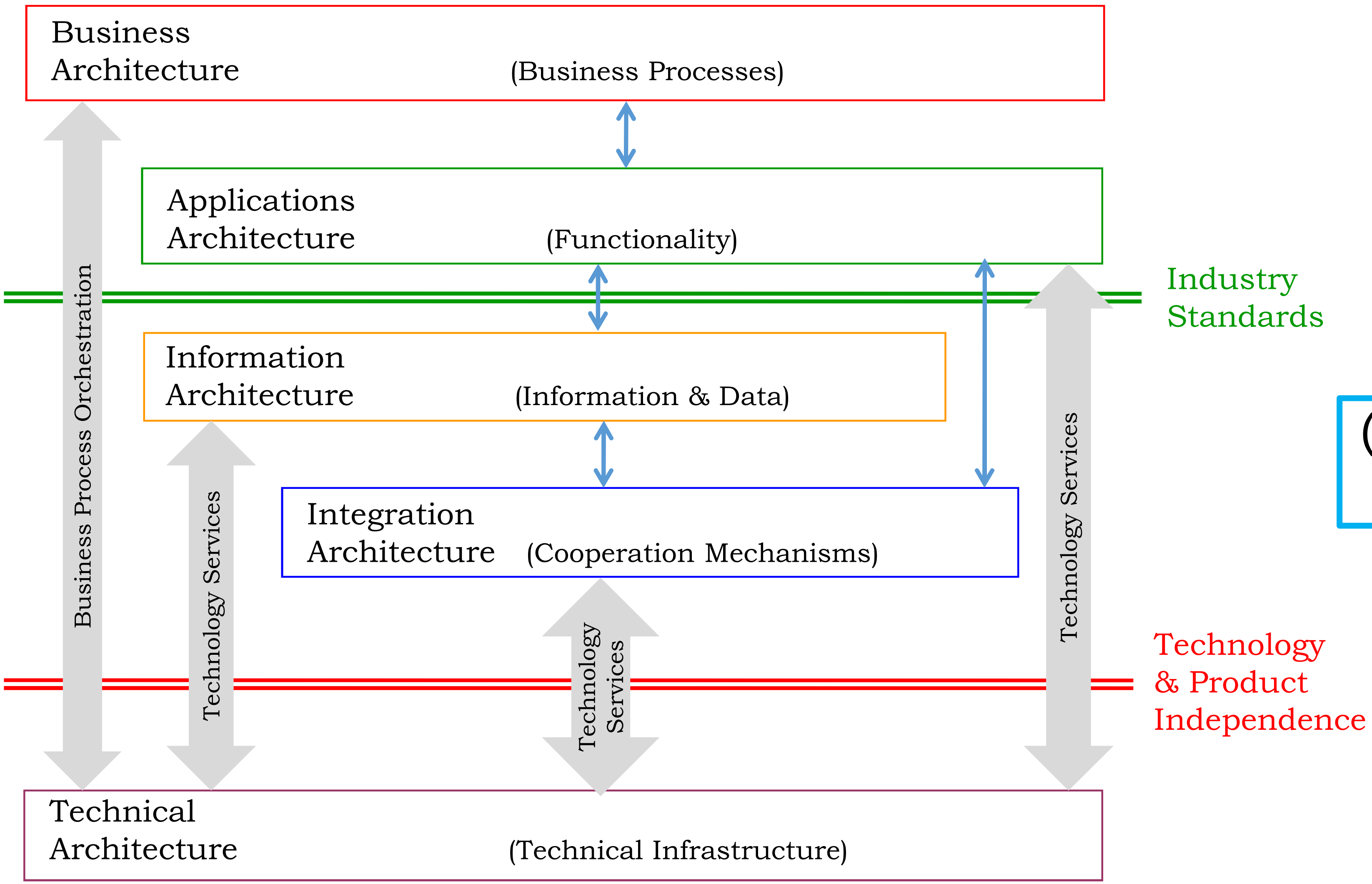
Misplacing **Vertical** Functionality



> 5'000 Applications

Misplacing **Vertical** Functionality





Horizontal Layer Interoperability

(Formalized) Services

Which are the mechanisms for layer-isolation ?

- Interlayer-access standards
  e.g. SQL-standard between technical infrastructure and applications

- Middleware
  Industry- or company-standard, stable middleware, such as an Enterprise Service Bus

- „Clean accesses“
  No use of vendor- or product-specific additions or enhancements

- Strict separation of layer functionality
  e.g. ***never*** implement technical functionality in the applications

- Strict, enforced programming guidelines
  e.g. explicitly allow/restrict/forbid certain programming constructs (Example: restrict stored procedures in DB-accesses)

- Open, long-term planning of the evolution of the infrastructure
  Evolution cycles with upward compatibility, well communicated

Business Architecture Layer

Applications Architecture Layer

Information Architecture Layer

Integration Architecture Layer

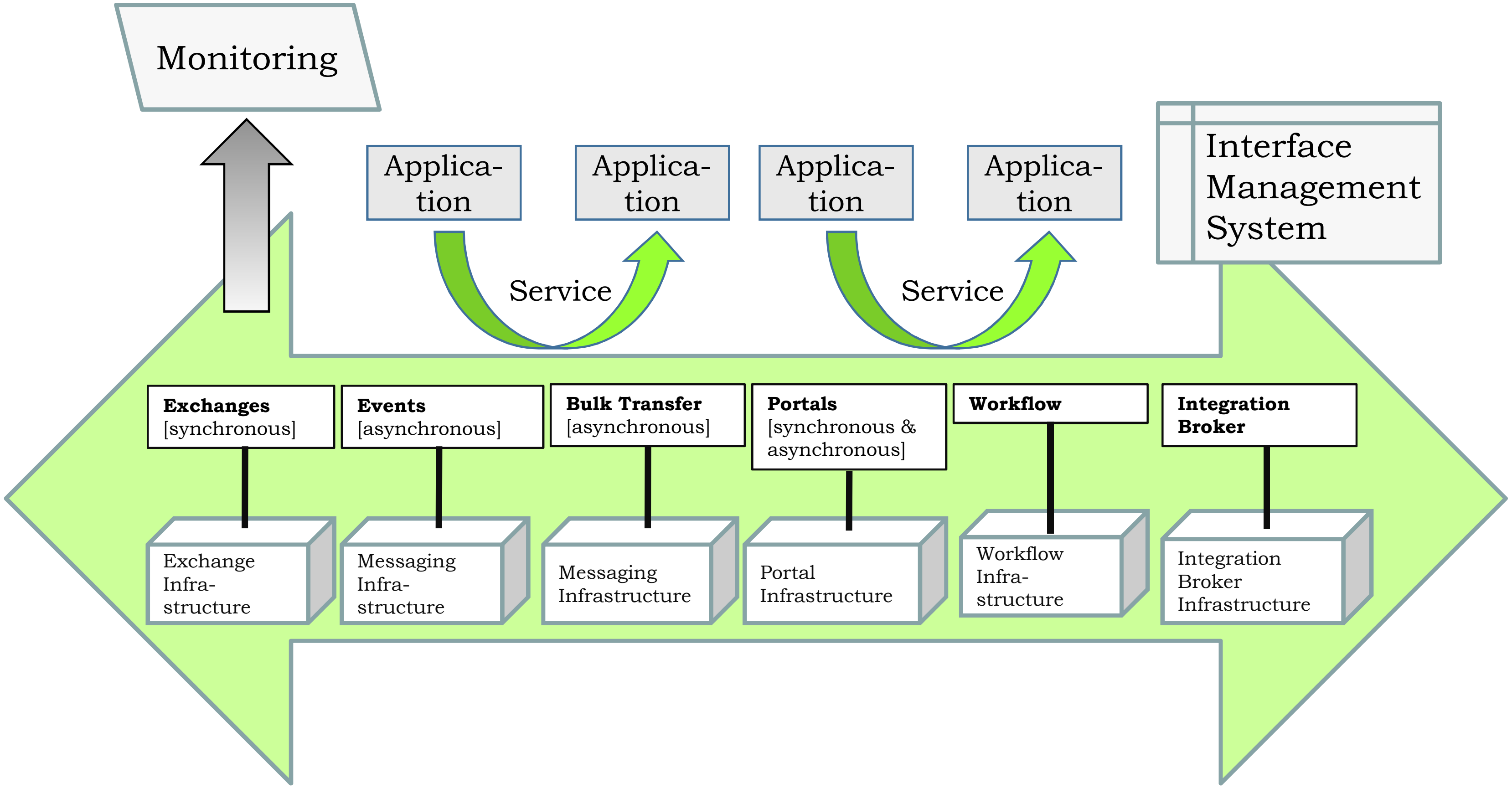
Technical Architecture Layer

Middleware: *provides the standardized infrastructure for the delivery of services in a distributed environment*

Additional Middleware Functionality:

- *Load balancing* (adaptive distribution of processing loads to servers)
- *Business continuity* (automatic mirroring of data and transactions)
- *Monitoring* (diagnostic and statistical information gathering, audit trail)
- *Security infrastructure* (access control to services, transport encryption)

Middleware Example: Enterprise Service Bus
(Large Company)



A1

Architecture Principle A1:
Architecture Layer Isolation

[1] Always use standardized, technology-independent, and product-independent mechanisms for transfer of data and control between layers

[2] Never implement functionality from vertical layers in the horizontal layers (especially no technical functionality in the applications)

Justification: Any reliance on specific technologies or product features generates dependencies which (massively) reduce changeability.

Architecture layers should be able to evolve in their own pace without impacting the other layers by force.

Vertical functionality should not be implemented in the applications (but accessed via services), otherwise changes impact the application landscape.

A2

Architecture Principle A2:

Partitioning, Encapsulation and Coupling

Horizontal Architecture Layer Principles:

- A1: Architecture Layer Isolation
- A2: Partitioning, Encapsulation and Coupling
- A3: Conceptual Integrity
- A4: Redundancy
- A5: Interoperability
- A6: Common Functions
- A7: Reference Architectures, Frameworks and Patterns
- A8: Reuse and Parametrization
- A9: Industry Standards
- A10: Information Architecture
- A11: Formal Modeling
- A12: Complexity and Simplification

«Spaghetti-Architecture»

https://i.ytimg.com

The Power of …

„The three devils of systems engineering are:
- Complexity,
- Change,
- Uncertainty"

Anonymous

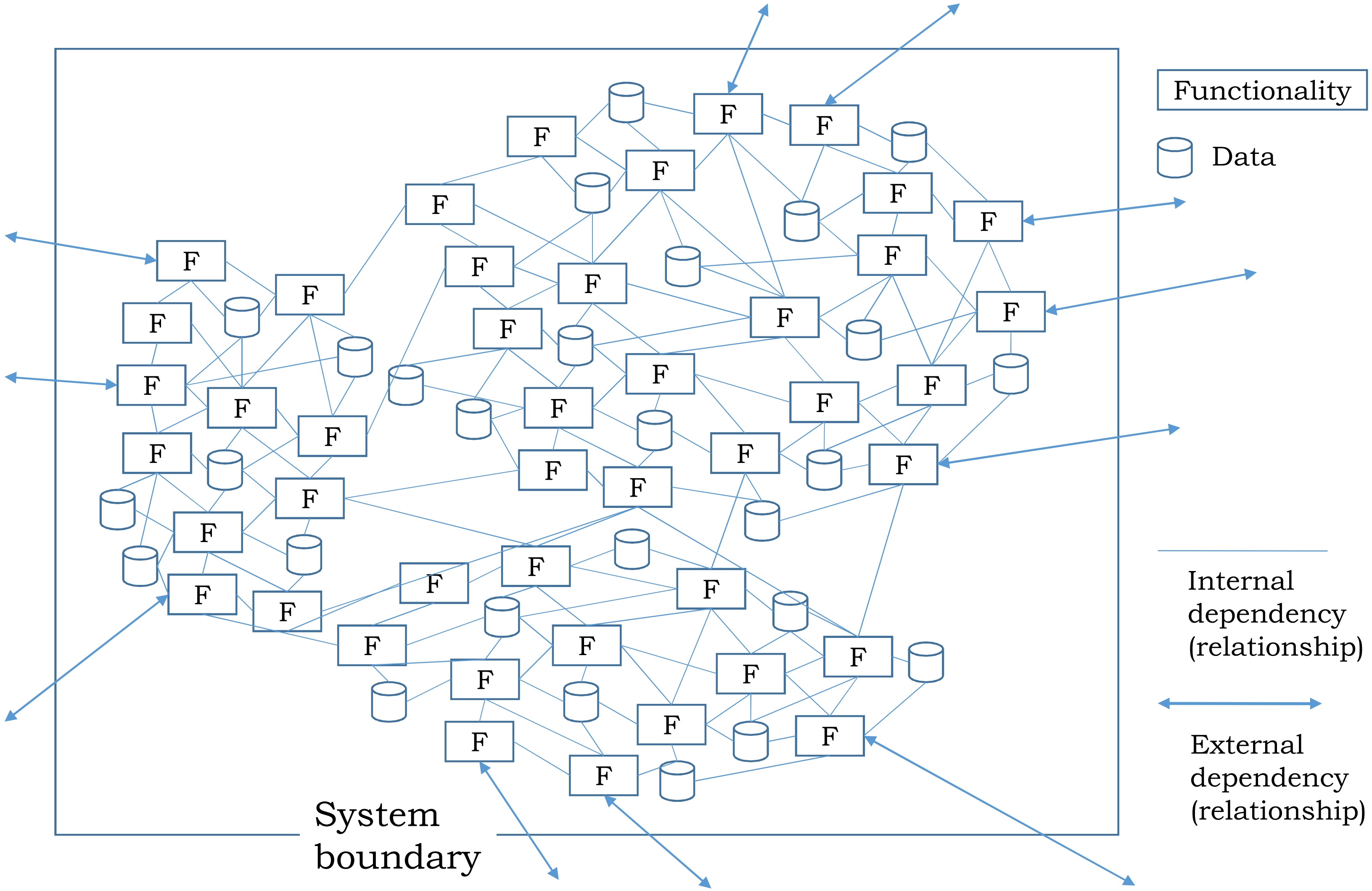
Therefore, *good engineering* is:

1. ***Reduce complexity*** as much as possible (Simplify)
2. ***Limit*** the effects and propagation of changes
3. ***Contain*** the risks of uncertainty

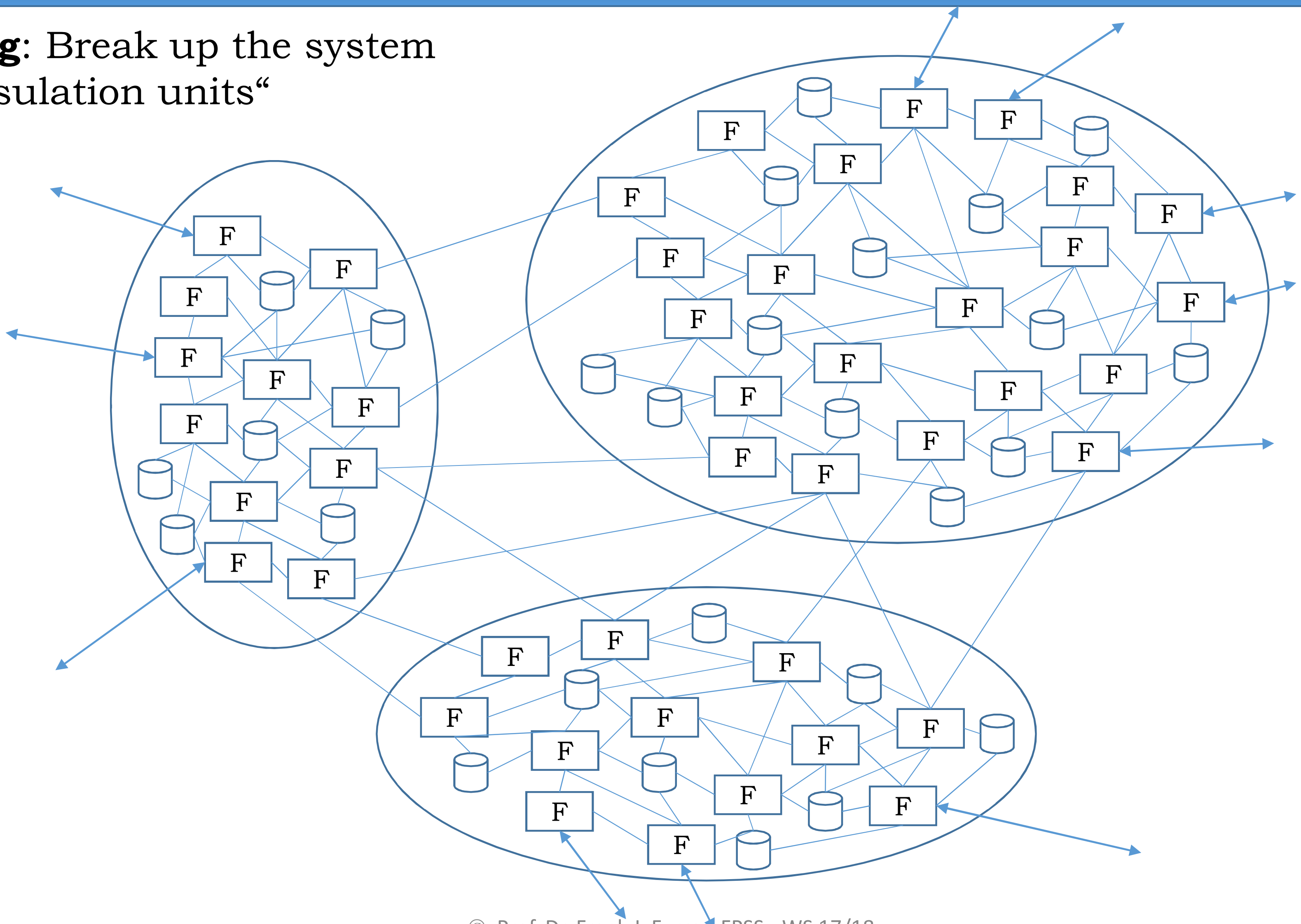
The most *powerful concepts* to do so are:

1. **Partitioning** of the system (→ smaller subsystems)
2. **Encapsulation** (→ hide the inner workings)
3. **Coupling** (→ stable interfaces and loose coupling)

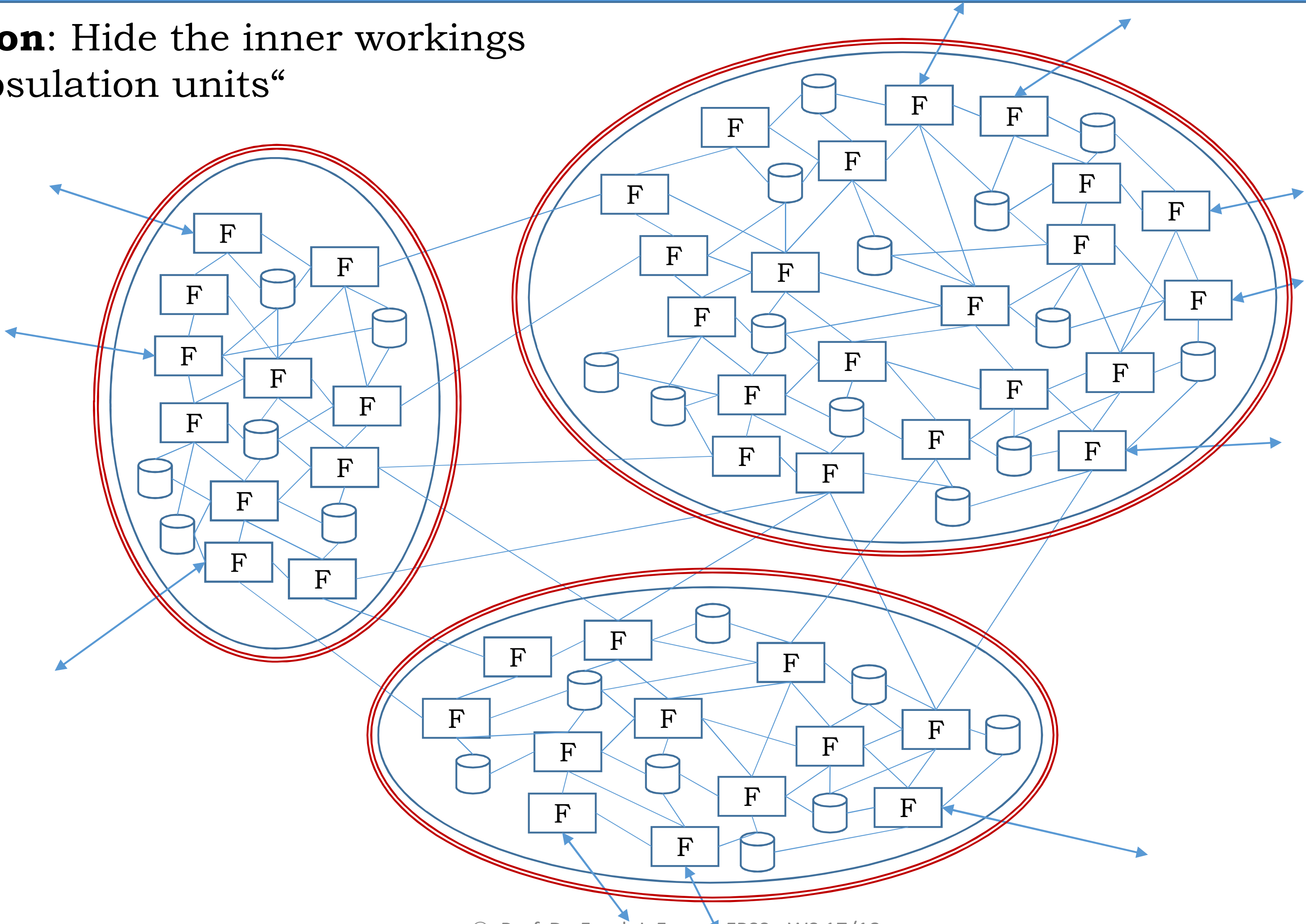
Definitions:



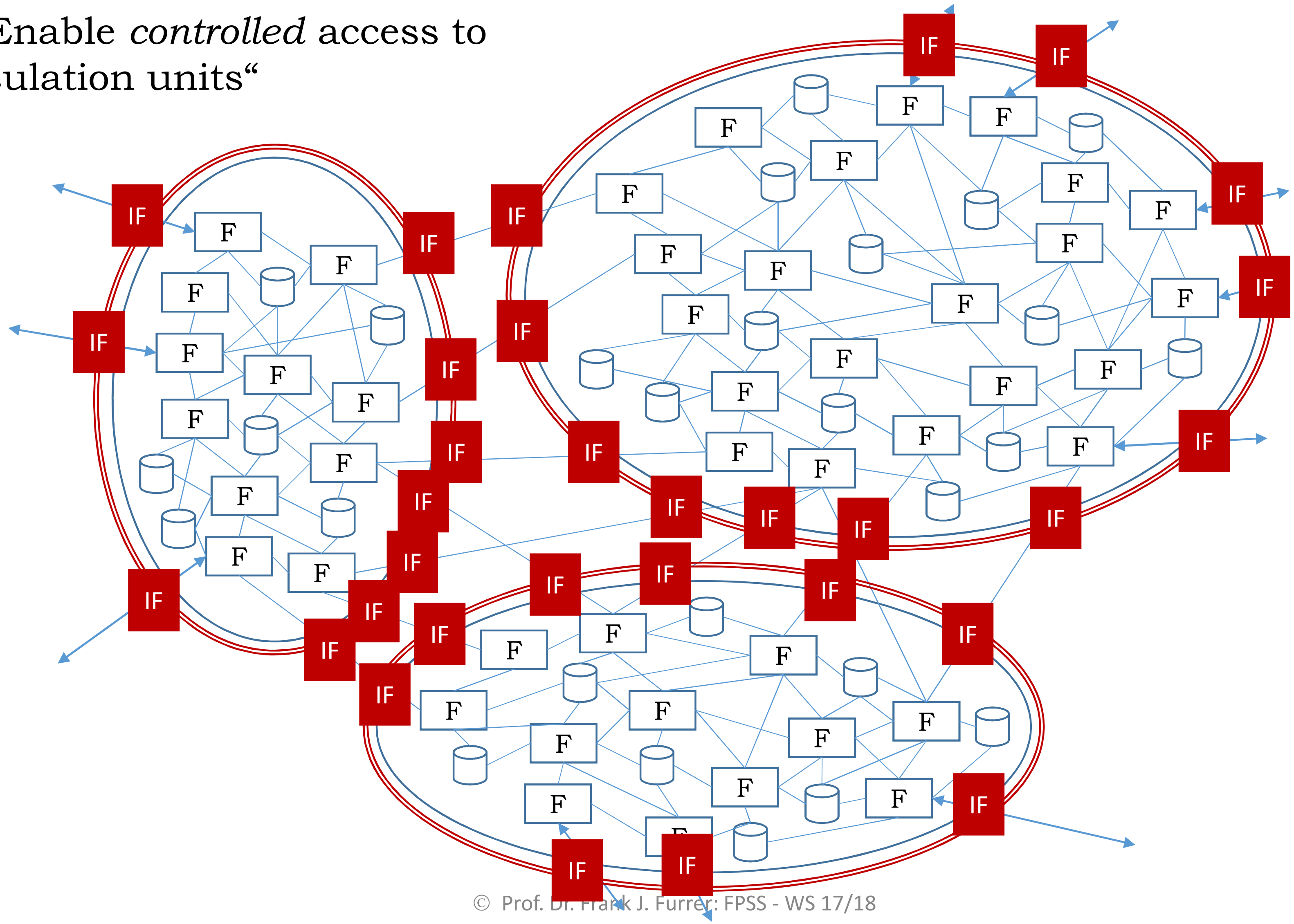
Partitioning: Break up the system into „encapsulation units“



Encapsulation: Hide the inner workings of the „encapsulation units“



Coupling: Enable *controlled* access to the „encapsulation units“



A2

Partitioning

Simplest possible **structure**:

- Highest cohesion
- Minimal redundancy
- Conceptual integrity

Encapsulation

Minimal **complexity**:

- Hiding
- Reduction to essentials
- Formalization

Coupling

Minimal **rigidity**:

- Weakest coupling
- Managed dependencies
- Contracts

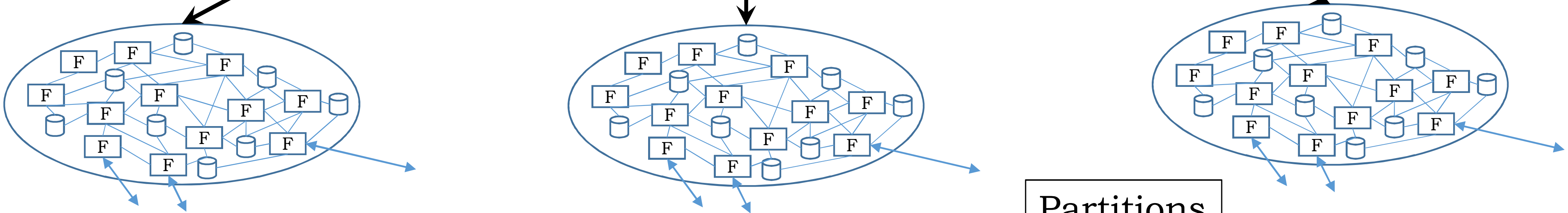
$\Rightarrow$ Changeability

Partitioning

http://img.fotocommunity.com

Functionality

Partitioning Rules



Partitions

Partitioning Rules

= Decision criteria for good partitioning

Primary Rule:
- Respect *cohesion*
- Avoid *redundancy*

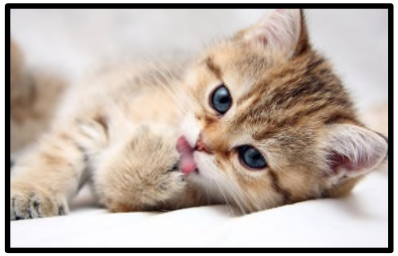
Other (secondary) criteria:
- Critical ⇔ non-critical (safety, confidentiality, …)
- Real-time ⇔ non real-time
- Rate of change: high ⇔ low
- Governance (Owner, stakeholder, country, …)
- Certified ⇔ uncertified

Respect *cohesion*

<u>Functional</u> ***Cohesion***: Assign functions to encapsulation units based on their cohesion

(„Dogs to Dogs, Cats to Cats“)


Partitions

<u>Data/Information</u> ***Cohesion***: Assign data and information to encapsulation units based on their similarity

(„Apples to Apples, Pears to Pears“)


Partitions


Example: Cohesion

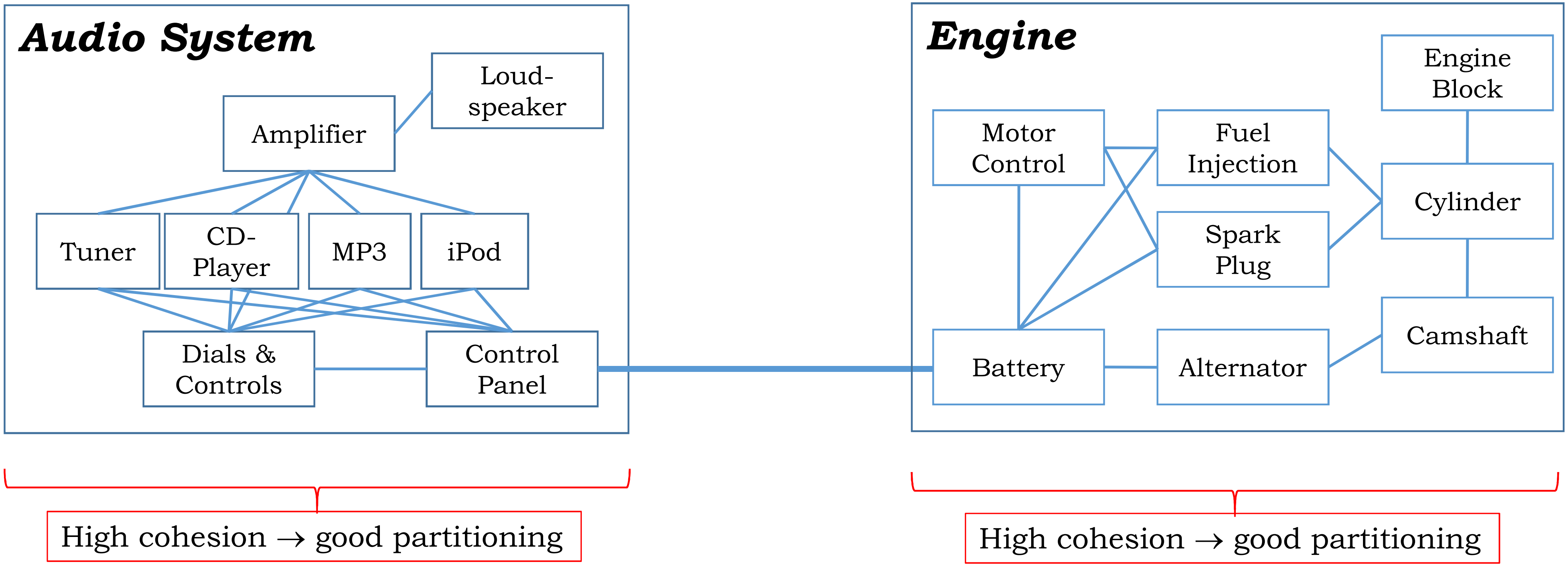
Functional and Data/Information ***Cohesion***: Assign functions and data to encapsulation units based on **least dependencies** (= *weak cohesion*)

Strong Cohesion:
- Team **A**
- Trained Cooperation
- Mutual Understanding
- Common Objective

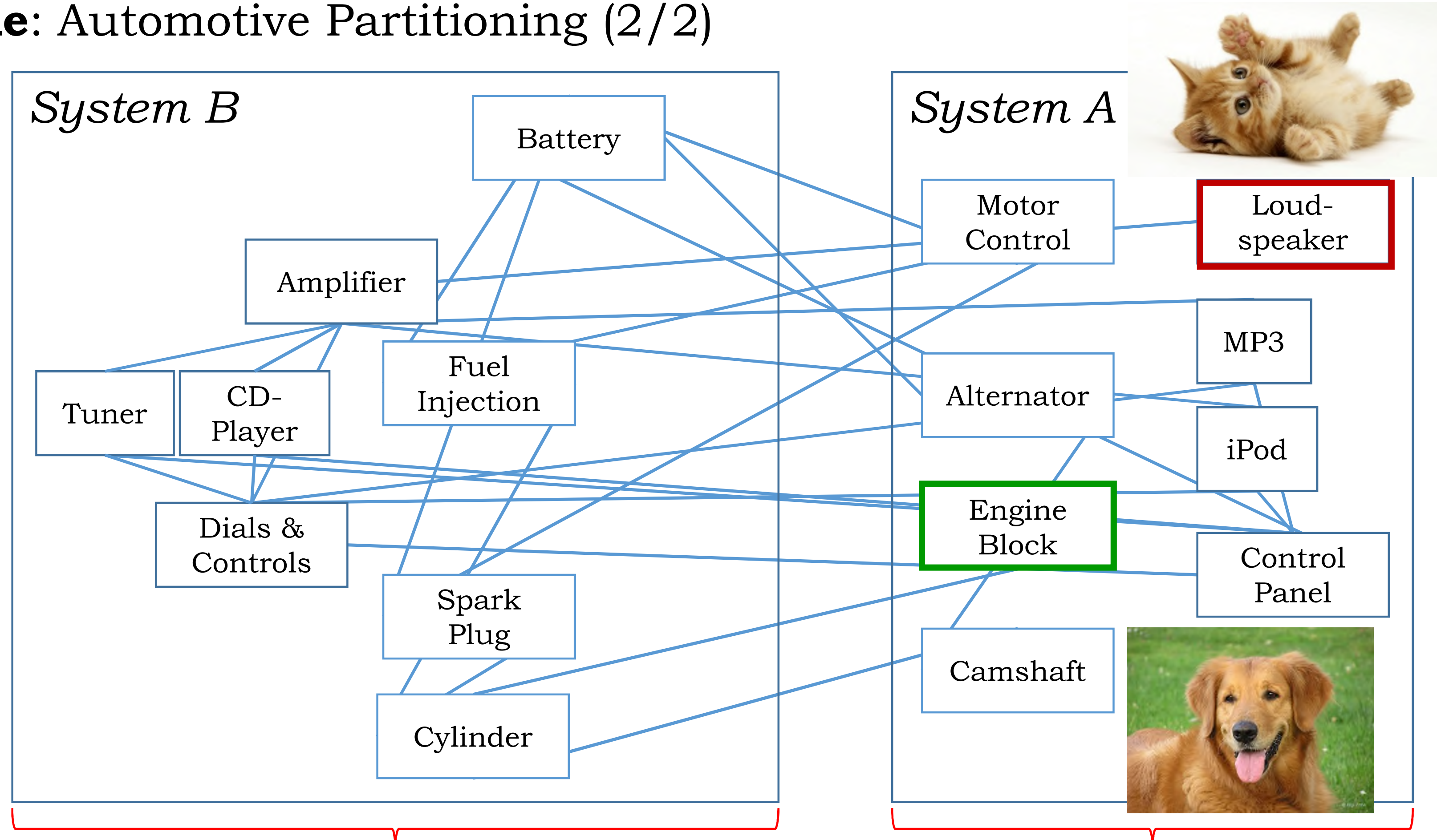
Weak Cohesion:
- Ball Contact

Strong Cohesion:
- Team **B**
- Trained Cooperation
- Mutual Understanding
- Common Objective

Example: Automotive Partitioning (1/2)



Example: Automotive Partitioning (2/2)



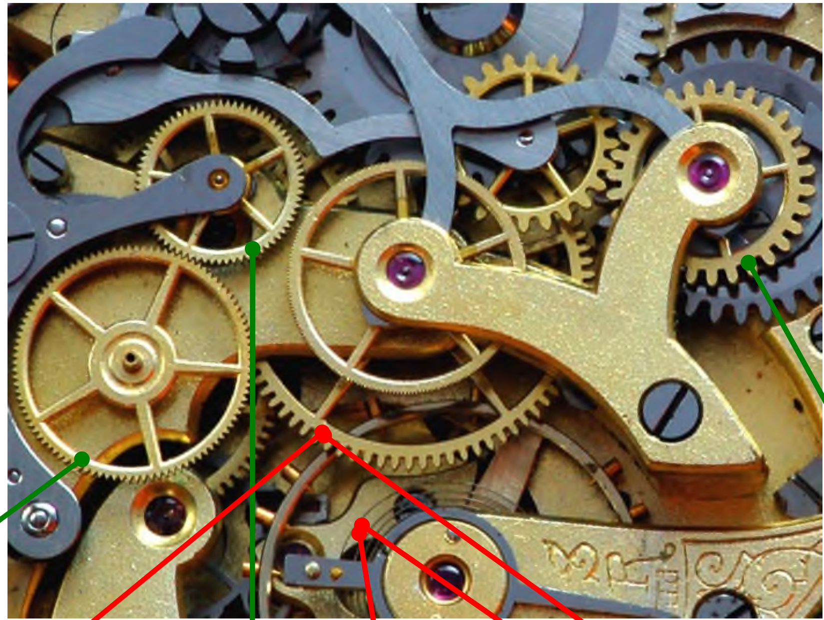
Low cohesion → bad partitioning

Low cohesion → bad partitioning

Primary Rule:

- Respect *cohesion*
- Avoid *redundancy*

Functional redundancy

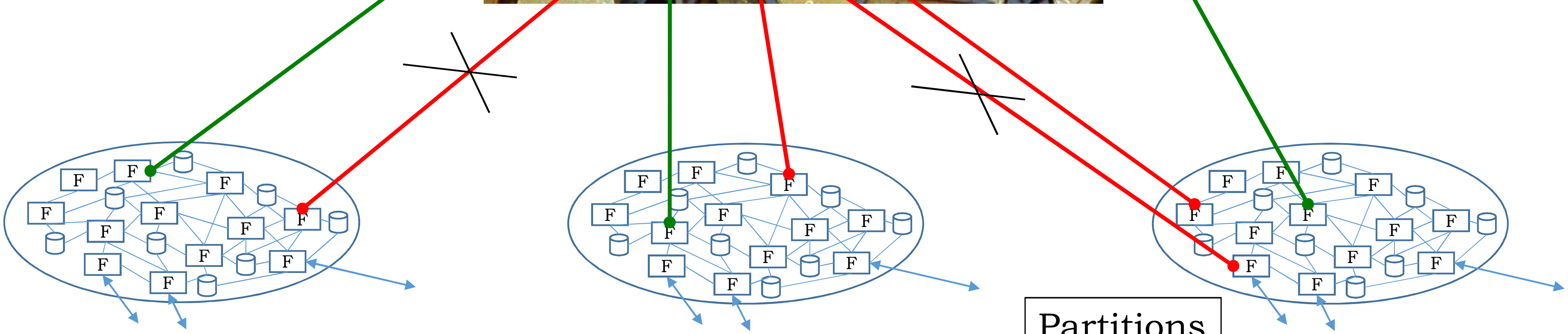


http://images.fotocommunity.de

GOLDEN RULE

A4

Redundancy!



Primary Rule:

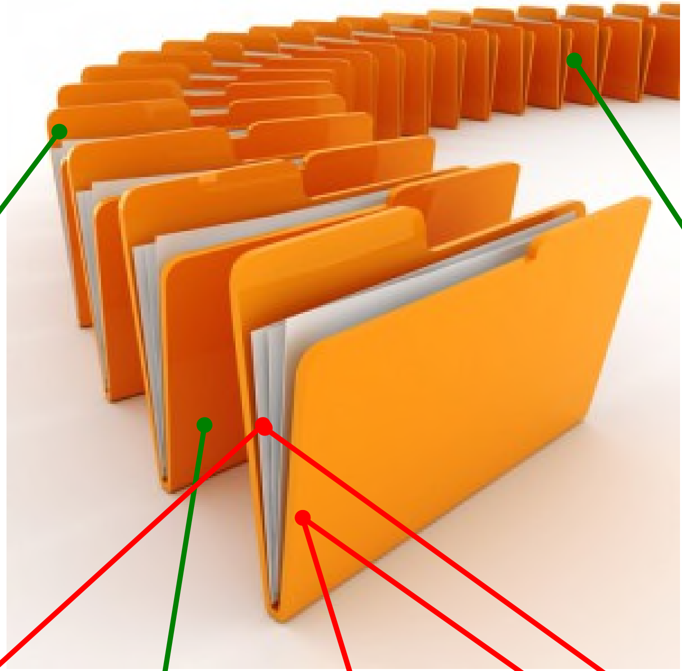
- Respect *cohesion*
- Avoid *redundancy*

Data redundancy

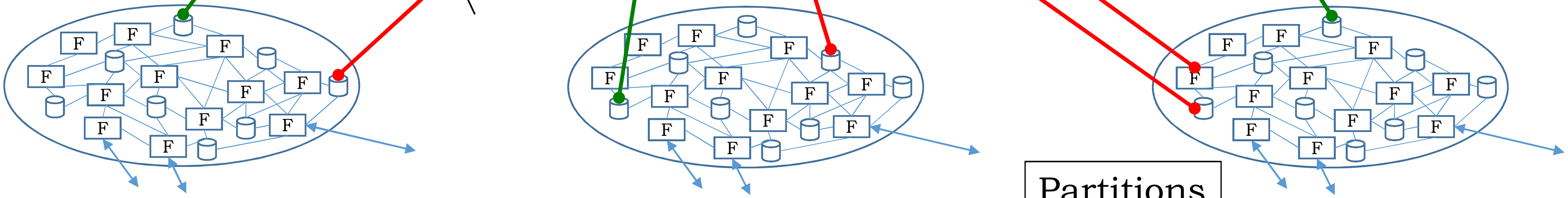
GOLDEN RULE

A4

Redundancy!

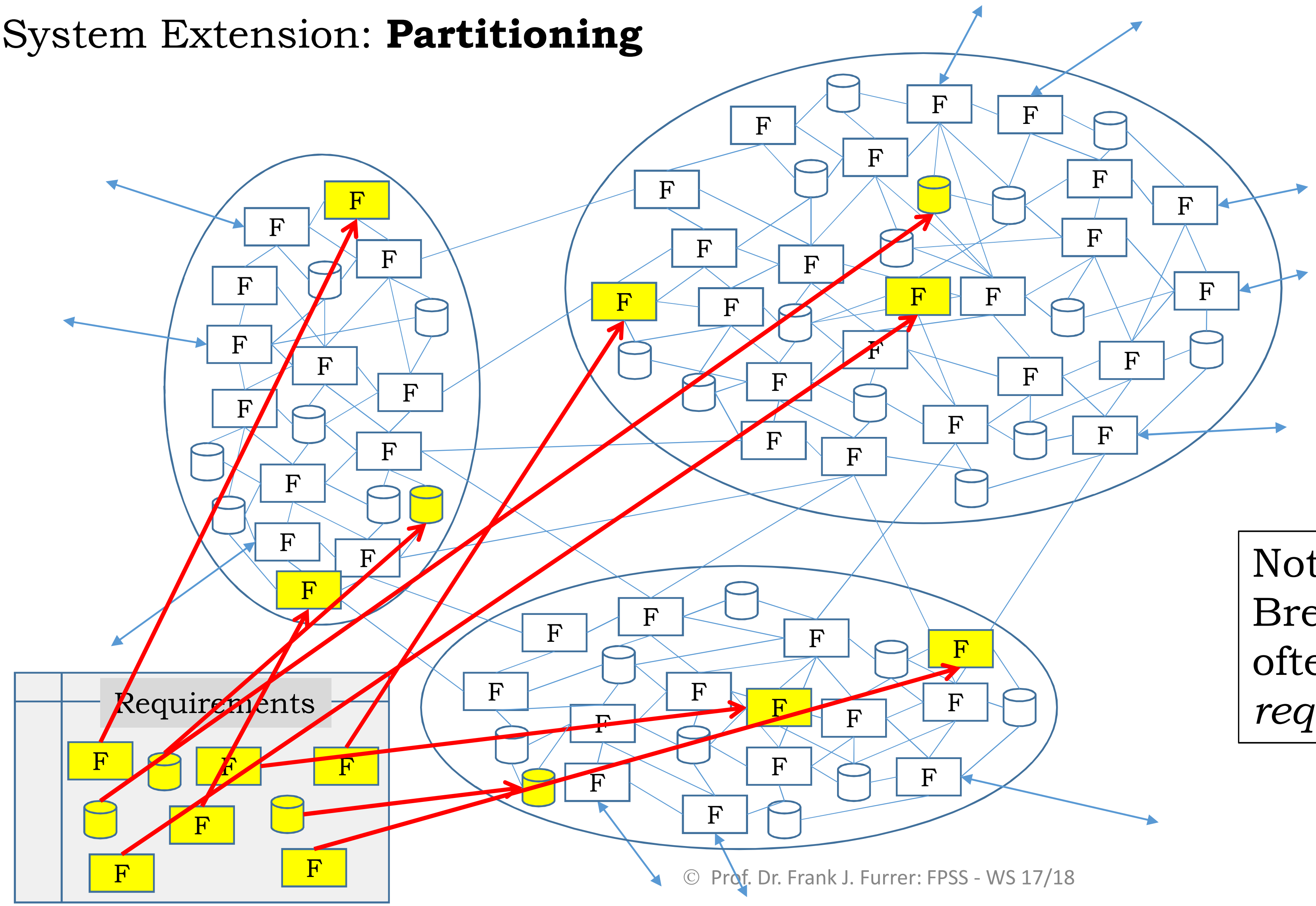


http://blogs.teradata.com



Partitions

System Extension: **Partitioning**

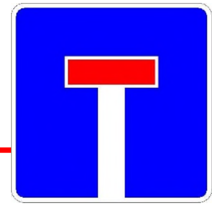


Note:
Breaking partitions often occurs during *requirements* phase

Partitioning Rules for Extensions

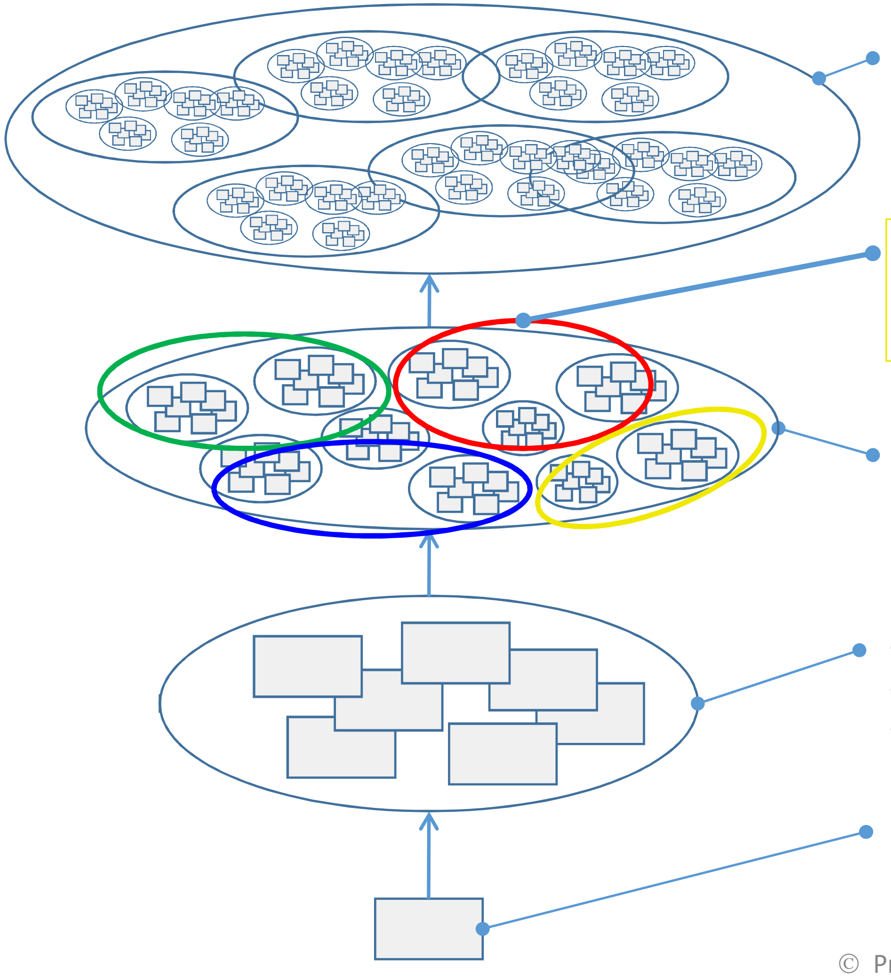
1. Assign the new functionality and data to *existing* encapsulation units according to their cohesion, i.e. respecting their „natural“ relationships („Apples to apples, pears to pears“)
2. Assign the functionality and data to *existing* encapsulation units according to their cohesion, i.e. minimize the number of external dependencies – especially tightly coupled dependencies
3. Create a new encapsulation unit whenever the required functionality or new data does not fit into the existing architecture
4. Keep functionality with the following properties:
   - Critical ⇔ non-critical (safety, confidentiality, …)
   - Real-time ⇔ non real-time
   - Rate of change: high ⇔ low
   - Governance (Owner, stakeholder, country, …)
   - Certified ⇔ uncertified
   - User interface layer ⇔ Process layer ⇔ Business logic layer
   - Specific functions & data ⇔ Common (X-system) functions & data

   in *separate* encapsulation units



WARNING:
Never implement functionality or data following the „least effort“ route!

Partitioning, Encapsulation and Coupling: **Domain Model**



System-of-Systems:
- Definition: Mission Engineer
- Concepts: Service Contracts

Domain Model:
- Definition: Business Engineer
- Concepts: Semantic Model

System:
- Definition: Software Architect
- Concepts: Interface Contracts

Component:
- Definition: Software Engineer
- Concepts: Interface Contracts

Module:
- Definition: Programmer
- Concepts: Programming Language Constructs

The domain model is a *strong instrument* to correctly assign functionality and data and to avoid unmanaged redundancy

Domain model:

Conceptual model defining the entities, their attributes, roles, relationships, and constraints that form an application domain

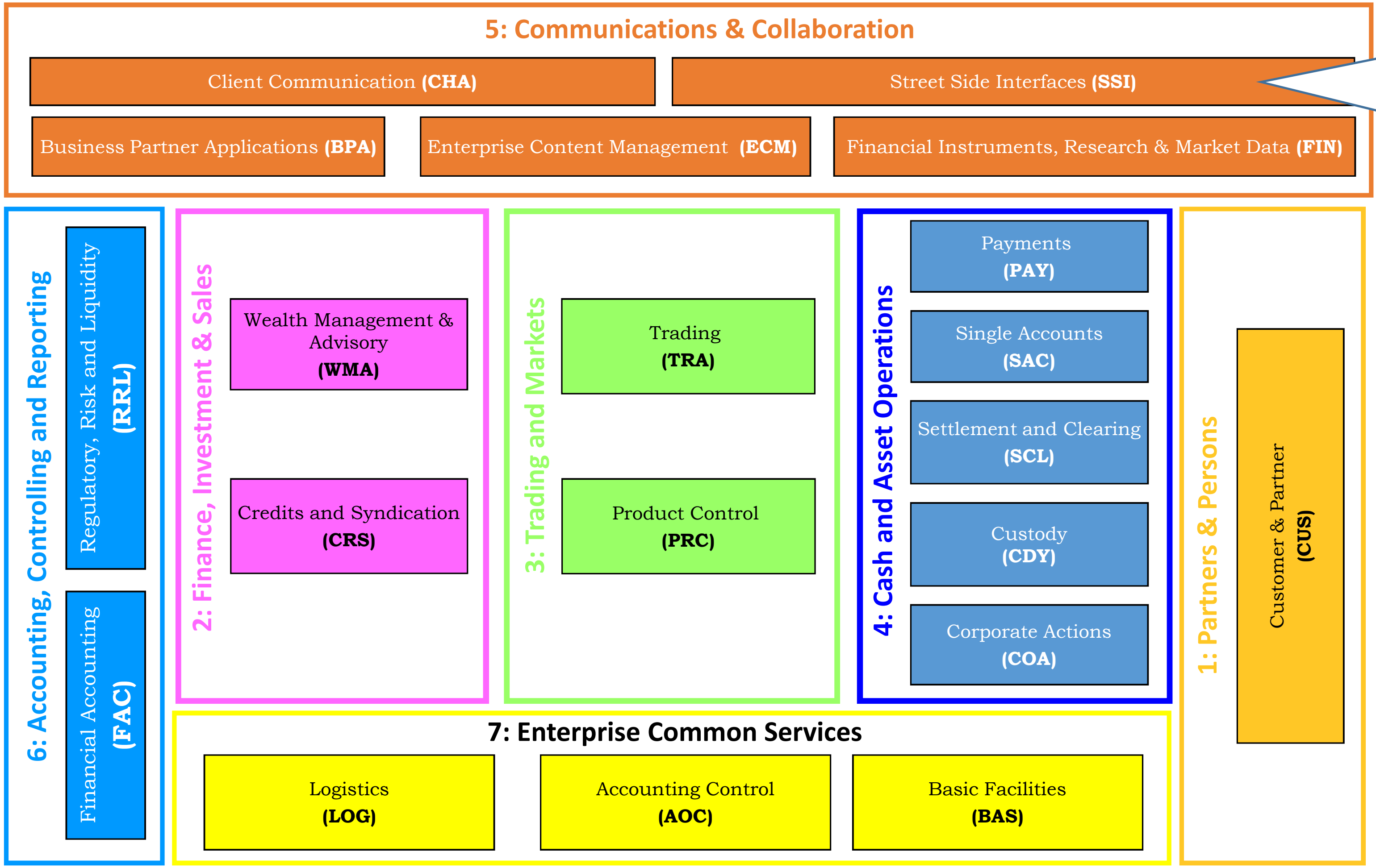
A domain model does not describe solutions to problems

The domain model represents the (business) concepts and their relationships in the application domain. A domain model shows a structural view of the domain.

A domain model is used to verify and validate the understanding of the application domain among various stakeholders (e.g. business ⇔ IT) and to document the evolution of the application domain

Example: Domain Model for a Financial Institution

This domain contains all applications for the communication with exchanges, clearing etc.



Example: Use of the Financial Institution Domain Model (1/3):

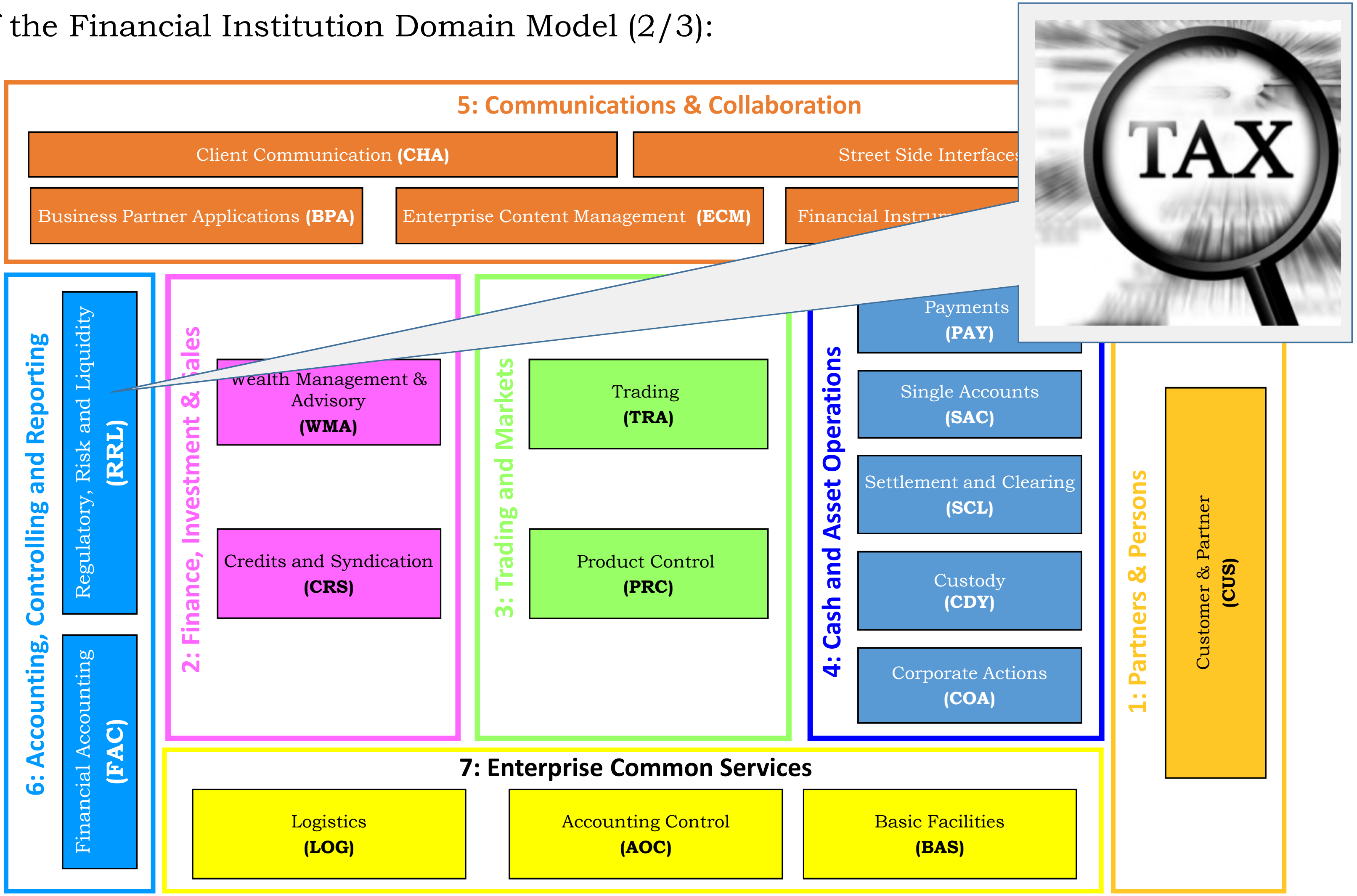
March 18, 2010:

http://www.nzz.ch/aktuell/wirtschaft/wirtschaftsnachrichten

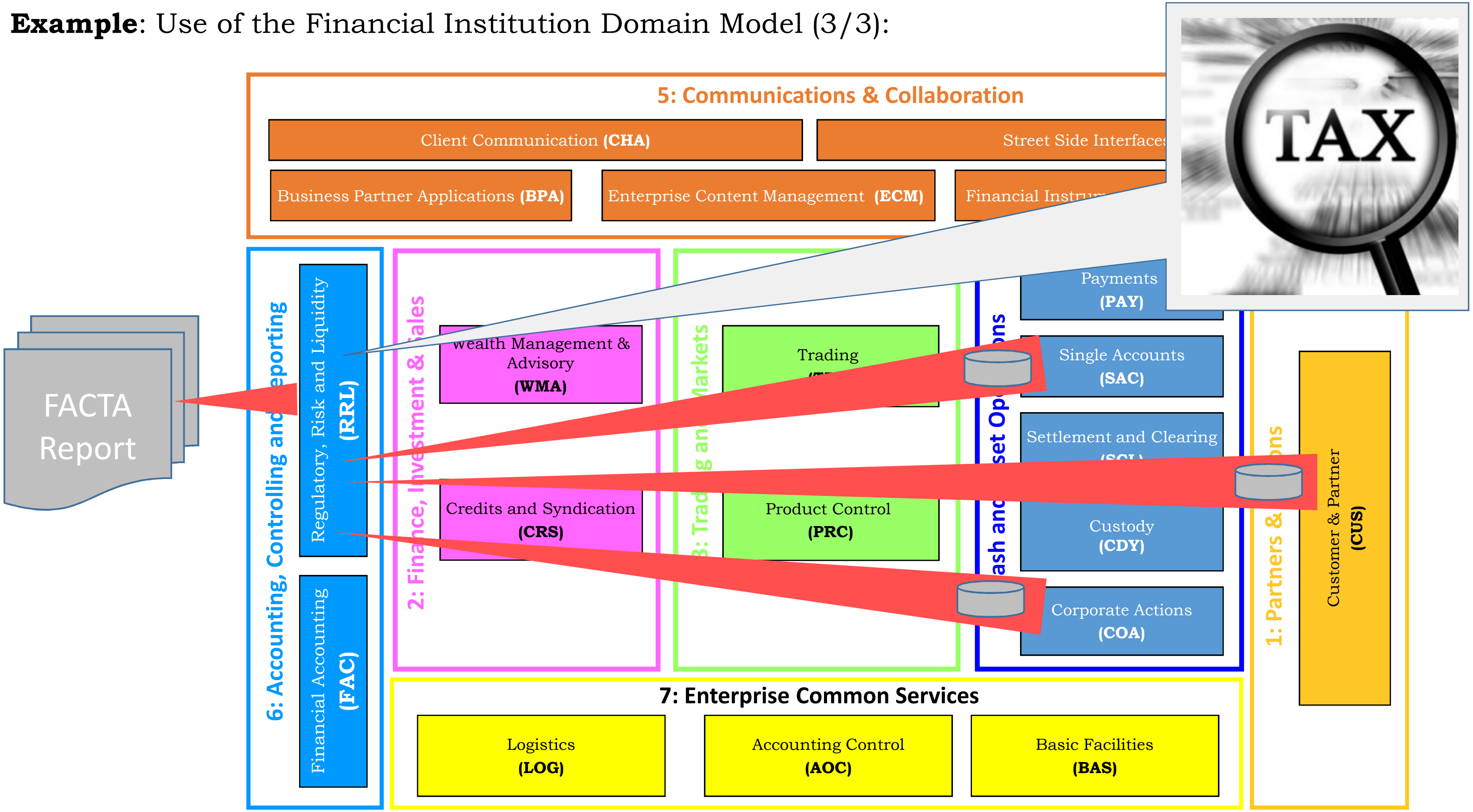
The **Foreign Account Tax Compliance Act** (**FATCA**) requires foreign financial institutions to report to the U.S. Internal Revenue Service (IRS) about their American clients (to combat offshore tax evasion)

Massive IT-Problem!

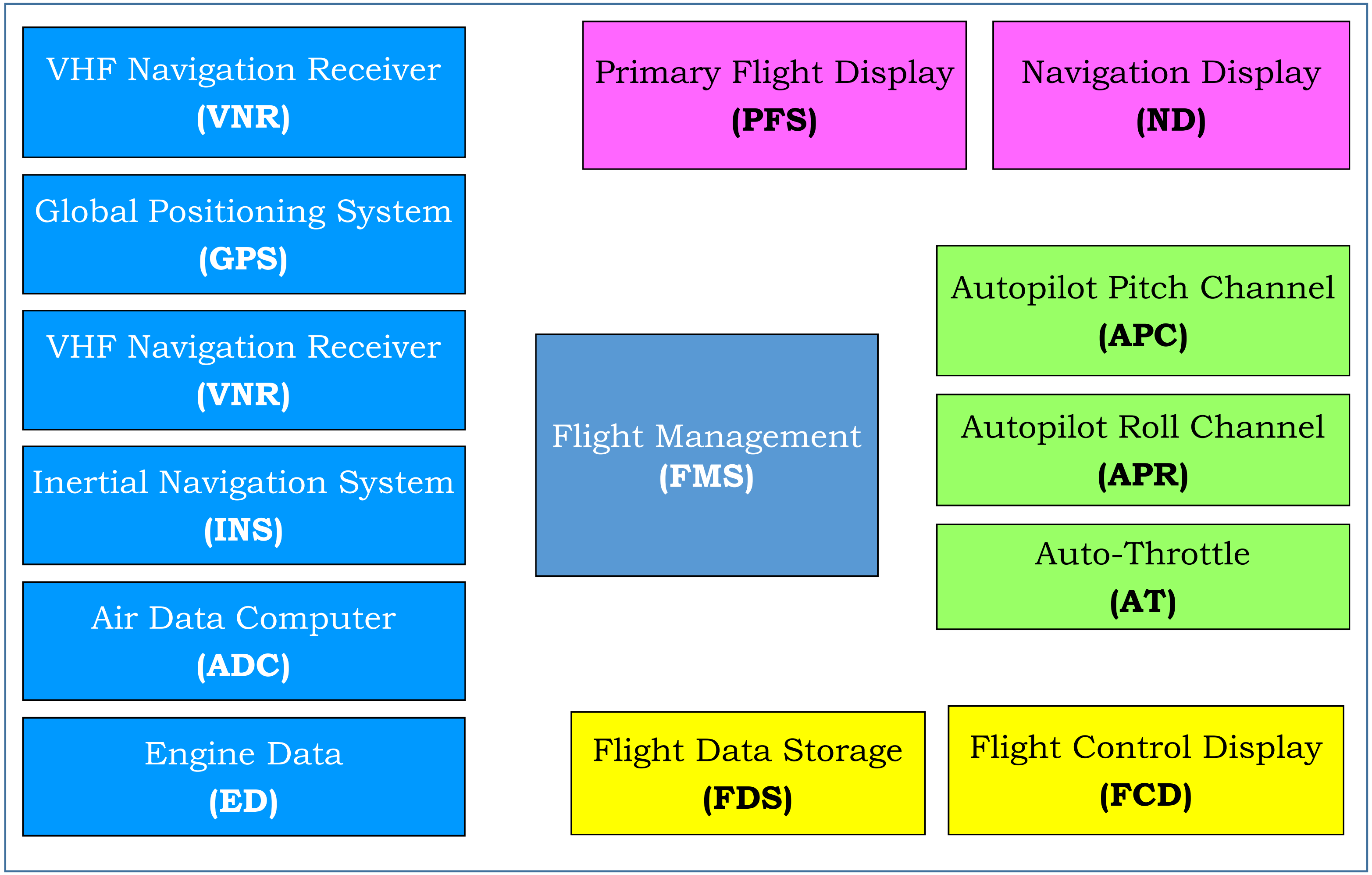
Example: Use of the Financial Institution Domain Model (2/3):



Example: Use of the Financial Institution Domain Model (3/3):



Example: Avionics Domain Model (1/3)



Example: Avionics Domain Model (2/3)

http://www.hyperraum.tv/tag/galileo-satellit/

Change Request:

U.S. Global Positioning System (**GPS**)

→ European Positioning System (**Galileo**)

Example: Avionics Domain Model (3/3)

http://www.hyperraum.tv/tag/galileo-satellit/

European Positioning System (**Galileo**)

VHF Navigation Receiver **(VNR)**

Global Positioning System **(GPS)**

VHF Navigation Receiver **(VNR)**

Inertial Navigation System **(INS)**

Air Data Computer **(ADC)**

Engine Data **(ED)**

Flight Management **(FMS)**

gation Display **(ND)**

Autopilot Pitch Channel **(APC)**

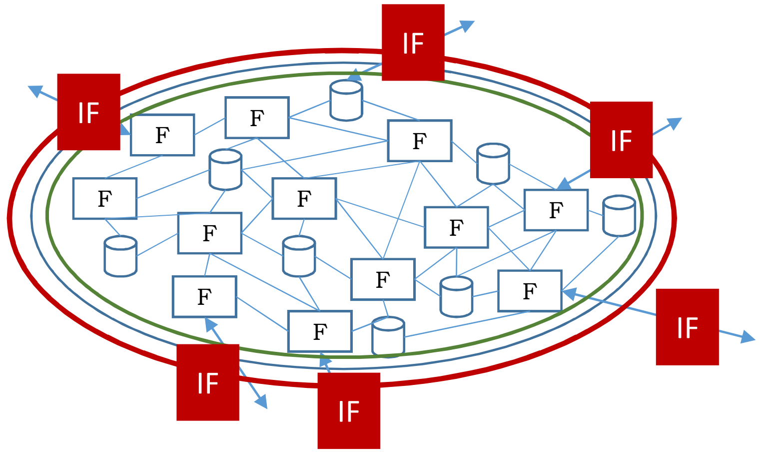
Autopilot Roll Channel **(APR)**

Auto-Throttle **(AT)**

Flight Data Storage **(FDS)**

Flight Control Display **(FCD)**

Encapsulation



http://schorsch.efi.fh-nuernberg.de

The inner workings of the encapsulation unit are hidden from the outside

All accesses are only allowed through well-specified (formally defined) *interfaces*

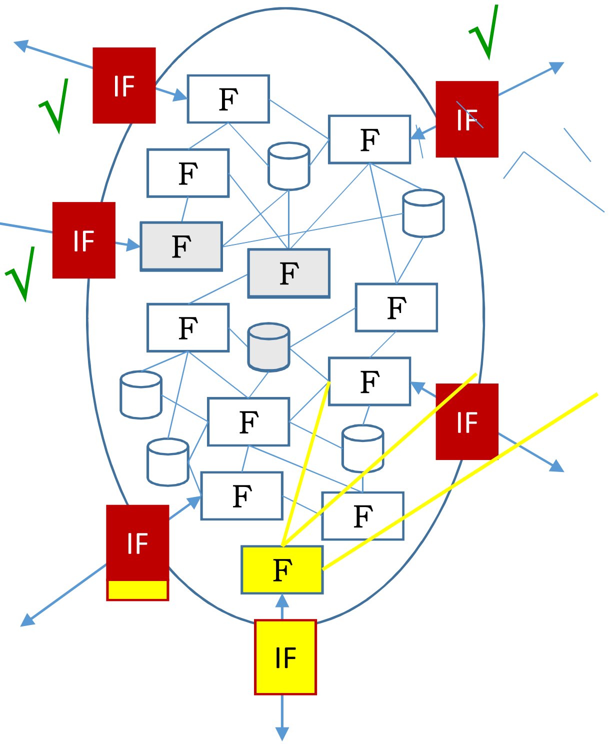
System Extension: **Encapsulation**

Refactoring
(New implementation or new technology)

Refactoring must not modify interfaces

Extension
(New functionality or new data)

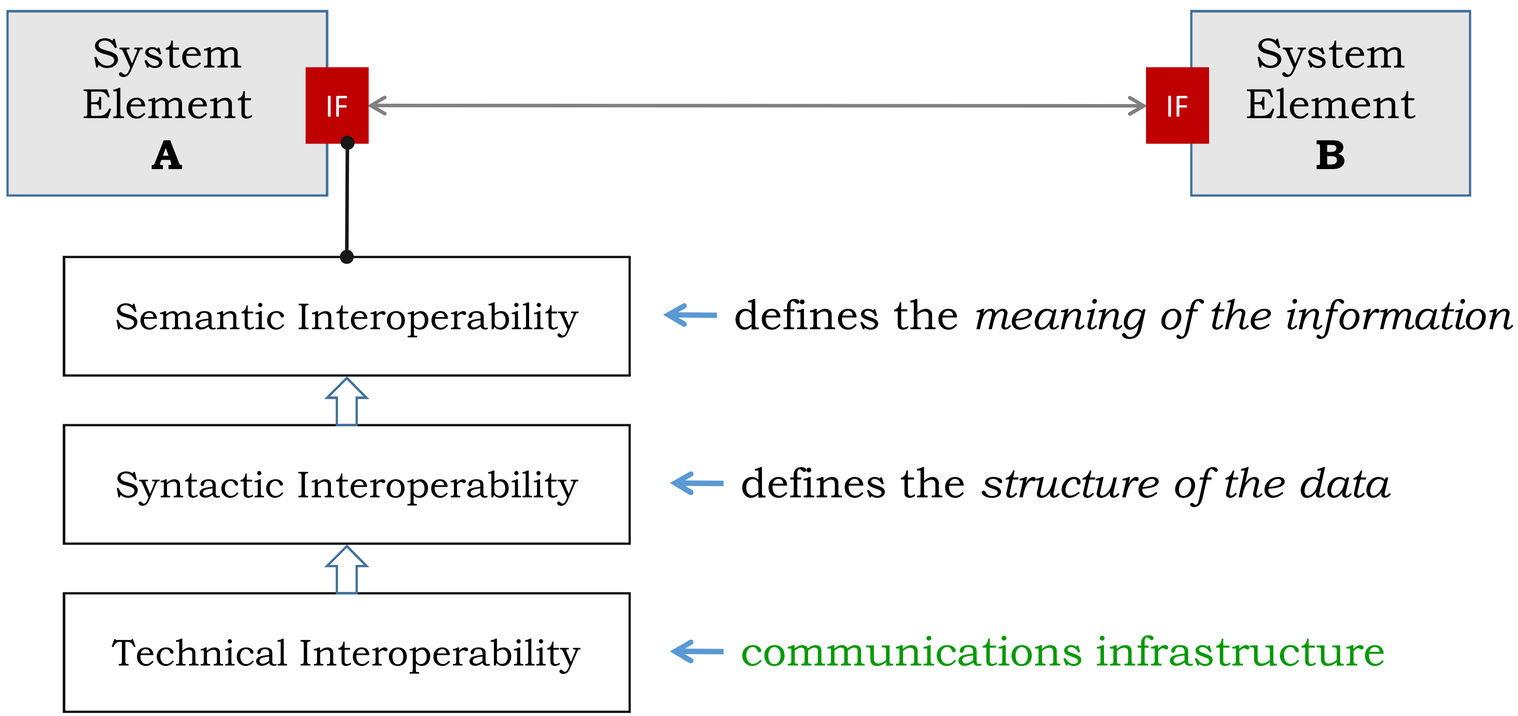
Extensions should leave interfaces unchanged, upwards-compatible or introduce new interfaces



System Extension: **Encapsulation**

Interfaces are strong instruments to maintain and improve changeability.
Interfaces are at the heart of loose coupling and reuse capability

A5



System **Extension**: *Encapsulation* Rules for Extensions

1. Keep the interfaces coherent, i.e. group only functionality and data into one interface which belong together (from a business point of view)
2. Maintain an adequate granularity of interfaces (in the number of functions offered)
3. Refactoring must not change the interfaces. Extensions should offer upwards-compatible or new interfaces – for both the *provider* and the *consumer* (beware of functional redundancy!)
4. Define the interfaces as precisely as possible, both syntactically and semantically (preferably by formal modeling and interface contracts)
5. Keep the interfaces technology-independent
6. Explicitly specify the context of interface use (preconditions, postconditions)
7. Carefully maintain an interface repository
8. Avoid duplication or overlap of interface-functionality between different interfaces (redundancy!)

WARNING:
Carefully control and review the introduction of new interfaces into your system !

Example: Syntactically and semantically upward-compatible interface

Precise meaning
(Ontology, taxonomy)

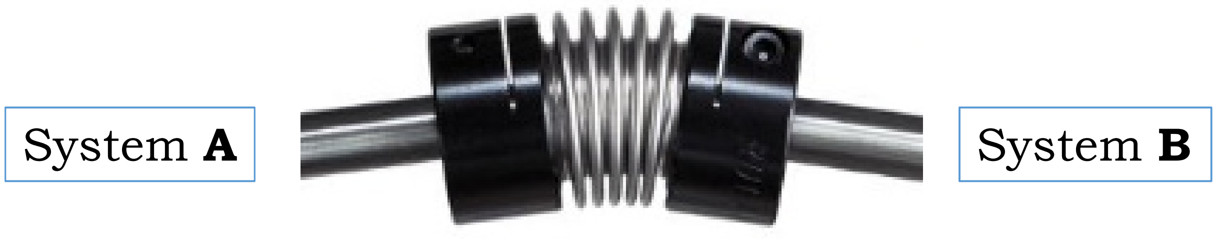
```
<xs:complexType name="RequestSlotType">
            <xs:sequence>
                        <xs:element name="SlotStatus"     type="dss:SlotStatus"/>
                        <xs:element name="BusinessUnit"   type="cs:businessUnitType"/>
                        <xs:element name="HashtreeUUID"   type="cs:uuid20AsCharType"/>
                        <!
                        <xs:element name="HashtreeRootValueLeft"        type="xs:base64Binary"/>
                        <xs:element name="HashtreeRootValueRight"       type="xs:base64Binary"/>
                        <xs:element name="HashtreeCheckLeftValueLeft"   type="xs:base64Binary"/>
                        <xs:element name="HashtreeCheckLeftValueRight"  type="xs:base64Binary"/>
                        <xs:element name="HashtreeCheckRightValueLeft"  type="xs:base64Binary"/>
                        <xs:element name="HashtreeCheckRightValueRight" type="xs:base64Binary"/>
            </xs:sequence
</xs:complexType>
```

precise XML-syntax

Upward-compatible interface extension:

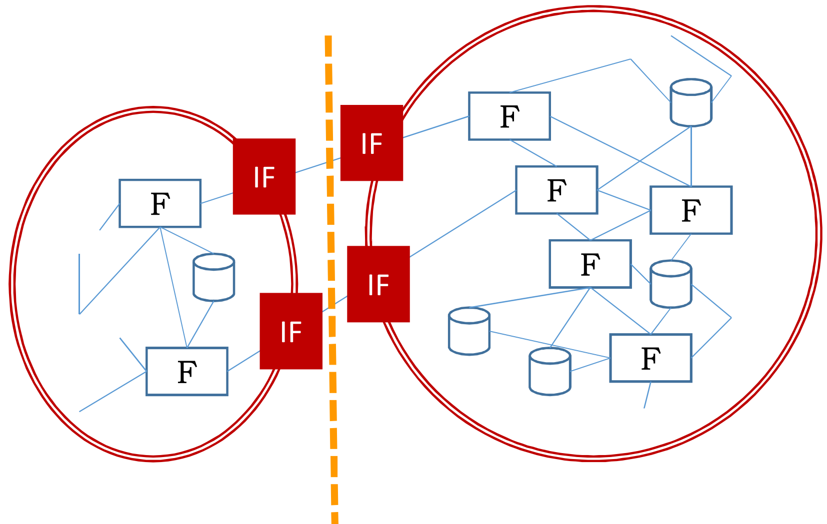
Any consumer which does not require the 2 new values *ignores* them and does **not** have to be changed!

System Extension: **Coupling**



Coupling causes *dependencies*:
- Functional dependencies
- Temporal dependencies
- Technical dependencies
- Semantic dependencies
- Operational dependencies

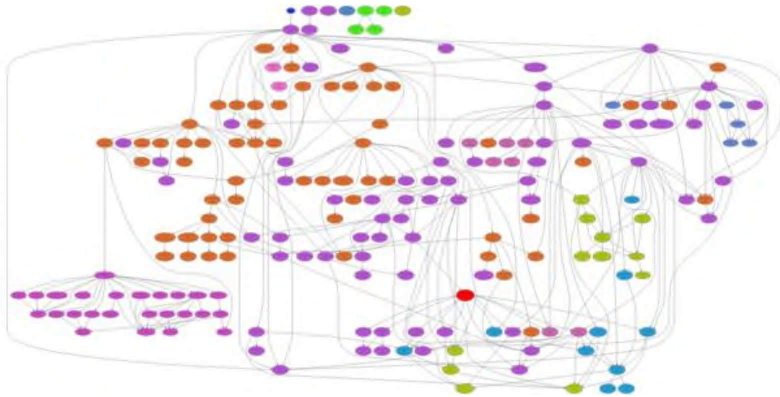
Dependencies exist both during *development* time and during *run-time*



Coupling:
Transfer of information or control

Coupling:

Each dependency *reduces* changeability



http://www.tarind.com/depgraph.html

Why?

- Any change in a system may impact the dependent systems and also force changes
- For any change in your system you may be constrained by the dependent systems

https://pbs.twimg.com

How to minimize dependencies:

- Functional dependencies: *Implementation-independent contracts with explicit context*
- Temporal dependencies: *Coupling as loosely as possible*
- Technical dependencies: *Isolate architecture layers (Architecture Principle)*
- Semantic dependencies: *Precisely align semantics (Conceptual Integrity, Ontology-matching)*
- Operational dependencies: *Explicitly identify and manage*

Example: Coupling mechanisms

Coupling mechanism	Functional dependency	Temporal dependency	Technical dependency	Semantic dependency	Operational dependency
Data sharing:					
Memory	high	medium	high	low	medium
DB	high	medium	high	low	medium
Synchronous:					
CORBA	n.a.	high	high	low	high
DCE/RPC	n.a.	high	high	low	high
COM/DCOM	high	high	high	low	high
Java/RMI	high	high	high	low	high
Asynchronous:					
Message passing (MQ)	n.a.	low	low	low	medium
Web services	low	low	low	low	medium
Time-slotted:					
TTA	n.a.	high	medium	low	medium
FlexRay	n.a.	high	medium	low	medium
Transaction Monitor:					
Java (Oracle)	high	high	high	medium	high
Java (Arjuna)	high	high	high	medium	high
IMS (IBM)	high	high	high	medium	high

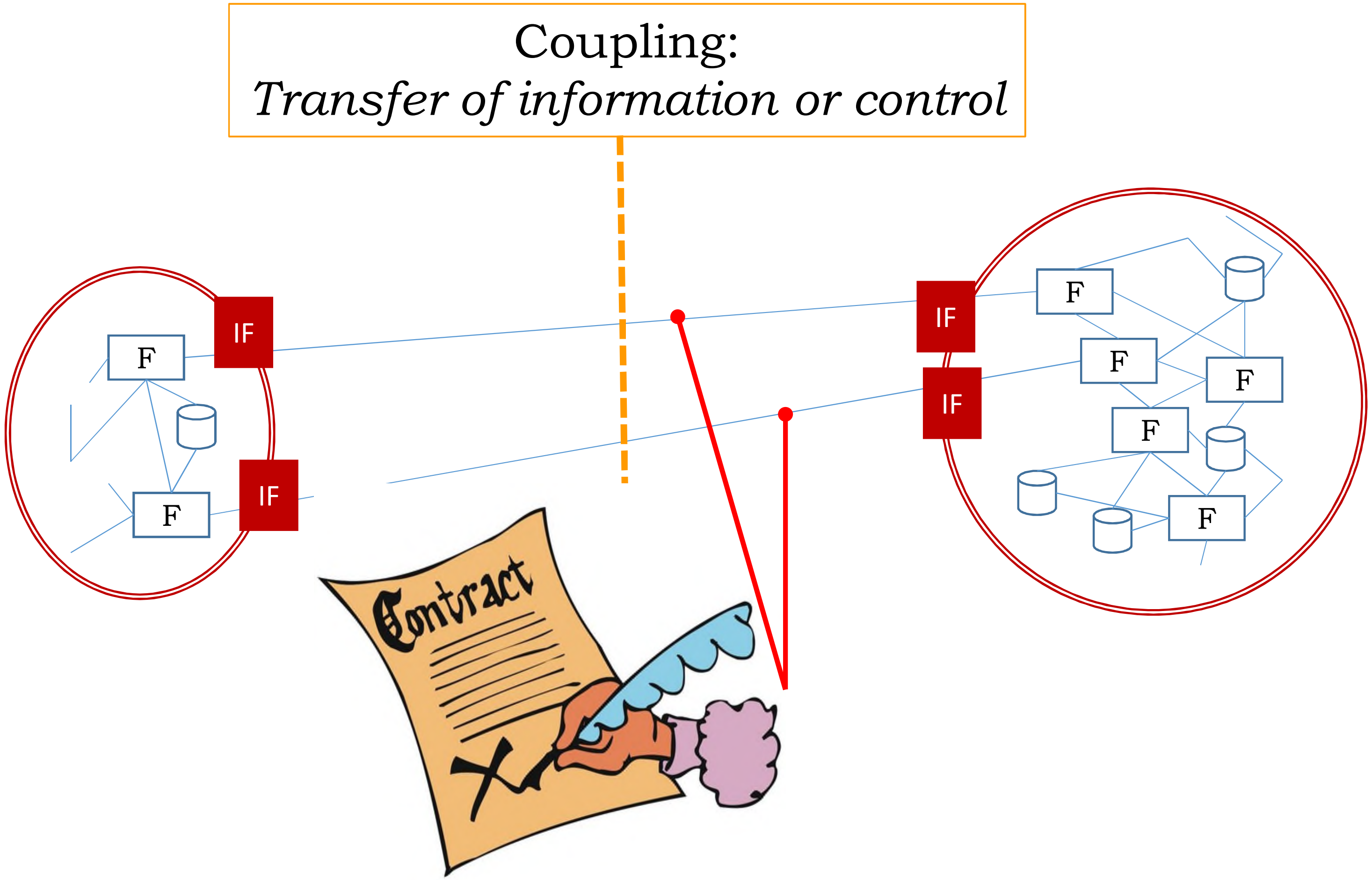
System **Extension**: *Coupling* Rules for Extensions

1. Minimize and standardize the number of coupling mechanisms which can be used by the developpers
2. Rely on standards, do not use product-specific features for coupling mechanisms
3. Couple as loosely as the application allows (preferably asynchronous, by message queuing)
4. Separate semantic issues as much as possible from the coupling mechanism

http://photaq.com

WARNING:
Beware of unnecessary tight coupling!

System Extension: **Coupling via Contracts**



Are there **other** partitioning rules?

Hierarchical Partitioning
(Dominant Quality Property)

Level 1

- Very High Performance
  - Level 2: HPC (High Performance Computing) → Reduced Architecture Principles
  - Level 2: Standard (Standard Performance Computing) → Full Architecture Principles
- Safety
  - Level 2: Safety-critical → Full Architecture Principles
  - Level 2: Non safety-critical → Full Architecture Principles

Level 2

http://www.brickbybrickinvesting.comm

A2

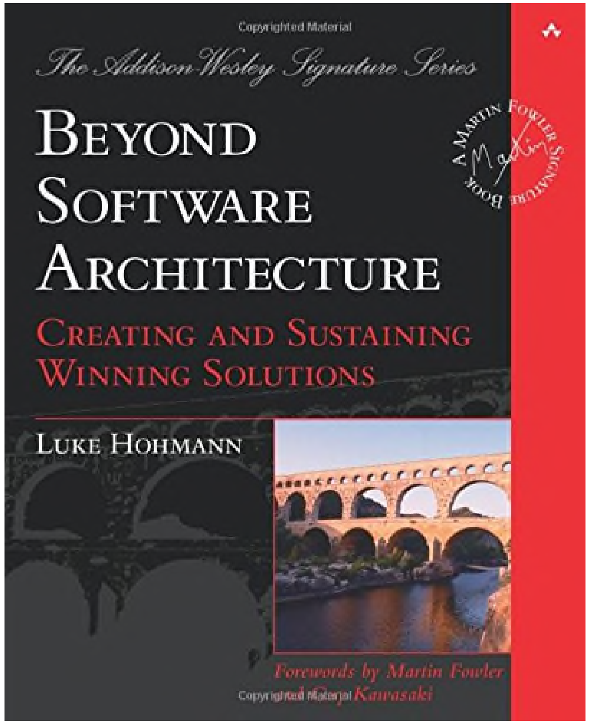
Architecture Principle A2:

Partitioning, Encapsulation & Coupling

1. Partition the functionality and data into encapsulation units according to their cohesion (thus minimizing dependencies)
2. Isolate the encapsulation units by strictly hiding any internal details. Allow access to functionality and data only through stable, well specified interfaces governed by contracts
3. Minimize the impact of dependencies between the encapsulation units by using adequate coupling mechanisms

Justification: These 3 rules minimize the number and the impact of dependencies. The resulting system therefore offers the least resistance to change, because any change affects the smallest possible number of system elements. A low resistance to change corresponds to high **changeability**.

Textbook



Luke Hohmann:
Beyond Software Architecture – *Creating and Sustaining Winning Solutions*
Addison-Wesley Professional, USA, 2003.
ISBN 978-0-201-77594-5

Textbook

Doug Kaye:
Loosely Coupled – *The Missing Pieces of Web Services*
RDS Press, California, USA, 2003. ISBN 978-1-881378-24-2

A3

Architecture Principle A3:

Conceptual Integrity

Horizontal Architecture Layer Principles:

- A1: Architecture Layer Isolation
- A2: Partitioning, Encapsulation and Coupling
- A3: Conceptual Integrity
- A4: Redundancy
- A5: Interoperability
- A6: Common Functions
- A7: Reference Architectures, Frameworks and Patterns
- A8: Reuse and Parametrization
- A9: Industry Standards
- A10: Information Architecture
- A11: Formal Modeling
- A12: Complexity and Simplification

How do we assure that all stakeholders have a *common and correct understanding*?

Wheel rotation sensor

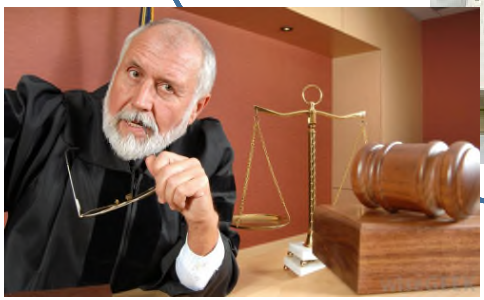
Book

Stock price

Car

Systems developpers

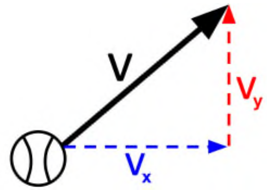
Software-System



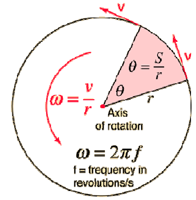
Legal system

Users

Velocity



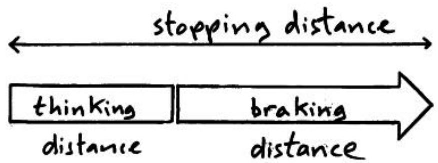
Rotation rate



Gravitation constant

$$F_g = G\frac{m_1 m_2}{r^2}$$

Stopping distance



https://thumb9.shutterstock.com

All systems are based on **concepts**

All systems use a **terminology**

All systems have **models** (implicit or explicit)

Divergence between stakeholders

Search Term	Google Search Results
Definition "system"	769’000’000
Definition «car"	321’000’000
Definition «velocity"	129’000’000
Definition «gravitational constant"	296’000

11.11.2017

https://lilpickmeupdotcom.files.wordpress.com

DEFINITIONS

Conceptual integrity is the quality of an organization and its IT-systems, where all the **concepts**, the **terminology** and the **models**, including their relationships with each other are unambiguously defined, applied and enforced in a consistent way

http://architecture.typepad.com/architecture_blog/2011/10/the-importance-of-conceptual-integrity.html

Conceptual integrity is the quality of an organization and its IT-systems, where all the **concepts**, the terminology and the models, including their relationships with each other are unambiguously defined, applied and enforced in a consistent way

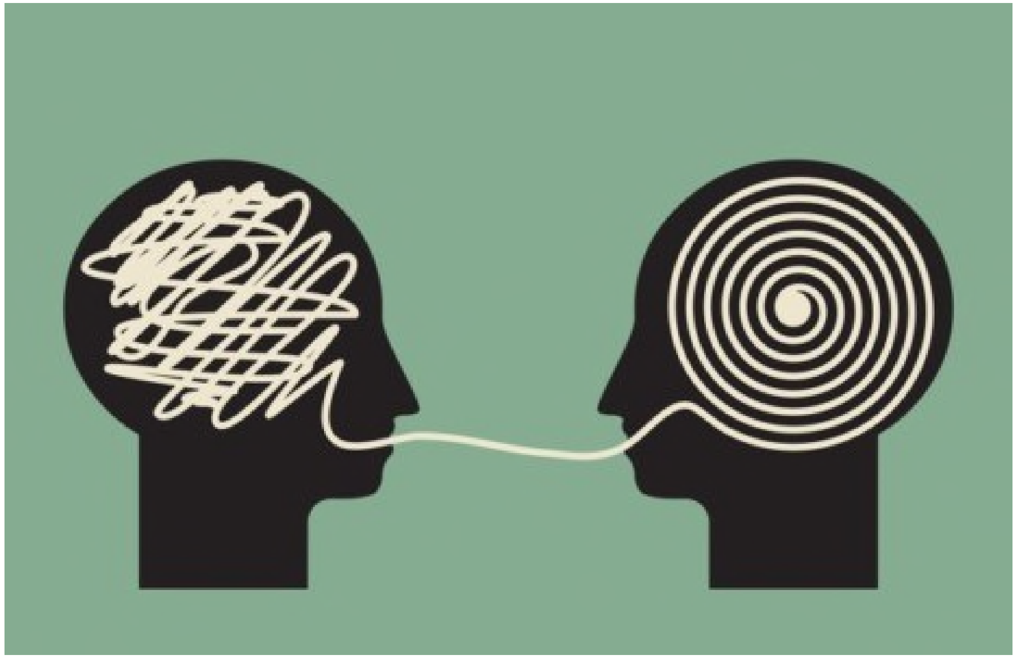
http://architecture.typepad.com/architecture_blog/2011/10/the-importance-of-conceptual-integrity.html

https://www.imagefactory.ch

DEFINITIONS

Concepts are the fundamental building blocks of our thoughts and beliefs. They play an important role in all aspects of cognition, communications and systems engineering.

https://en.wikipedia.org/wiki/Concept



In *cyber-physical systems*:
- Risk
- Accidents

https://i1.wp.com

http://dcnepalonline.com

Lack of *conceptual integrity* leads to:
- Misunderstandings of stakeholders
- Diverging implementations
- Unsatisfied users
- Unnecessary development and maintenance effort

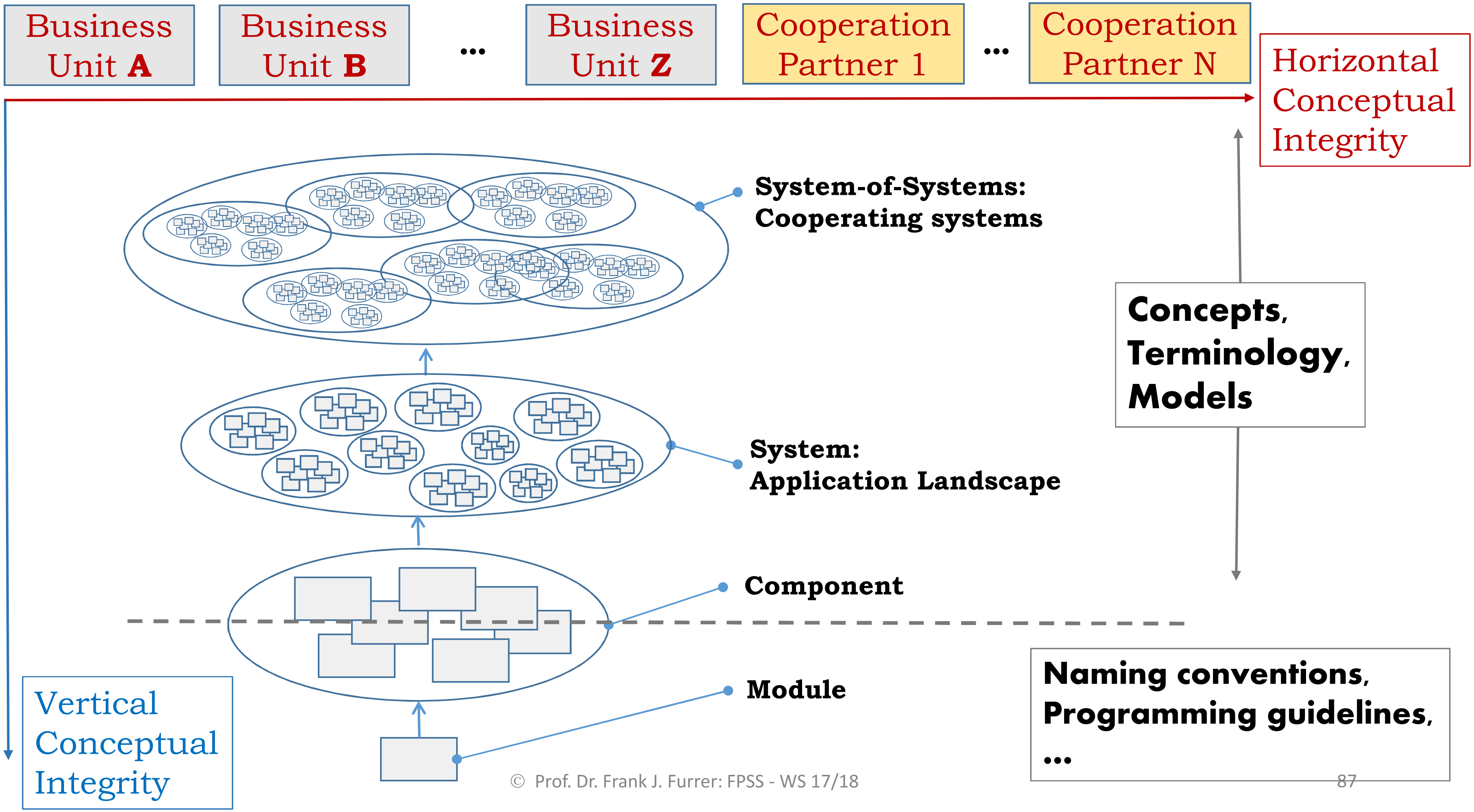
http://www.jubilee-centre.org

Creating, maintaining, and enforcing *conceptual integrity* is mandatory in IT systems

How can we ensure conceptual integrity?

… with a solid model foundation

- Taxonomy
- Ontology
- Domain model
- Business object model


Business Unit A
Business Unit B
...
Business Unit Z
Cooperation Partner 1
...
Cooperation Partner N
Horizontal Conceptual Integrity
System-of-Systems: Cooperating systems
Concepts, Terminology, Models
System: Application Landscape
Component
Module
Naming conventions, Programming guidelines, ...
Vertical Conceptual Integrity

Business

Conceptual Integrity **Definition**
- Terminology
- Domain Model Expertise
- Business Object Model Expertise

Review

Architecture

Conceptual Integrity **Formalization**
- Taxonomy/Ontology
- Domain Model
- Business Object Model

Conceptual Integrity Enforcement

Implementation

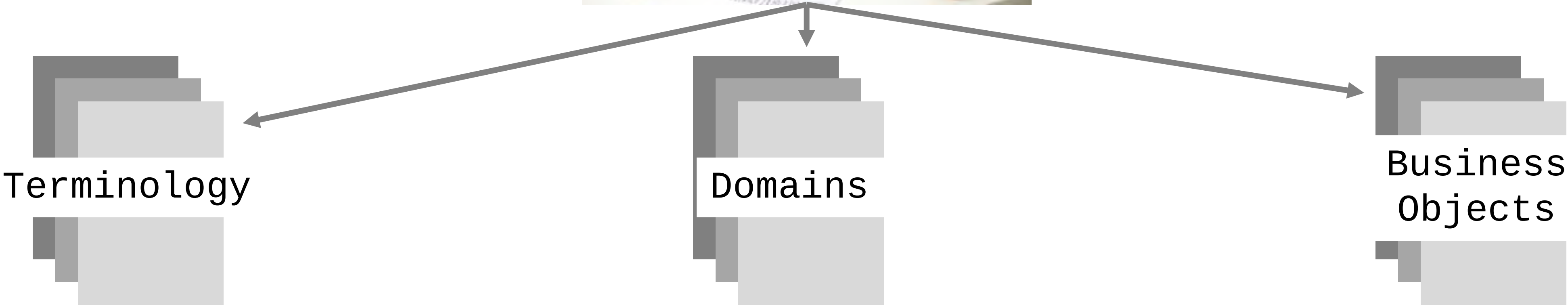
Conceptual Integrity **Implementation**
- Code
- Documentation

Business

Conceptual Integrity **Definition**

- Terminology
- Domain Model Expertise
- Business Object Model Expertise

Interviews
Workshops



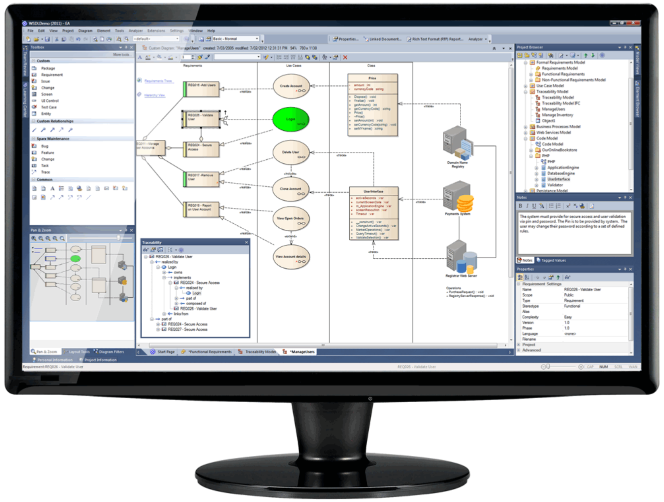
https://media.licdn.com

Architecture

Conceptual Integrity **Formalization**

- Taxonomy/Ontology
- Domain Model
- Business Object Model

Terminology Definition



Domain Model

Business Object Model

http://sparxsystems.com

Architecture

Domain Model

Domain Trading (TRA)

Trading **(TRA)**

Container for all functionality and data used in the **trading operations** of the bank

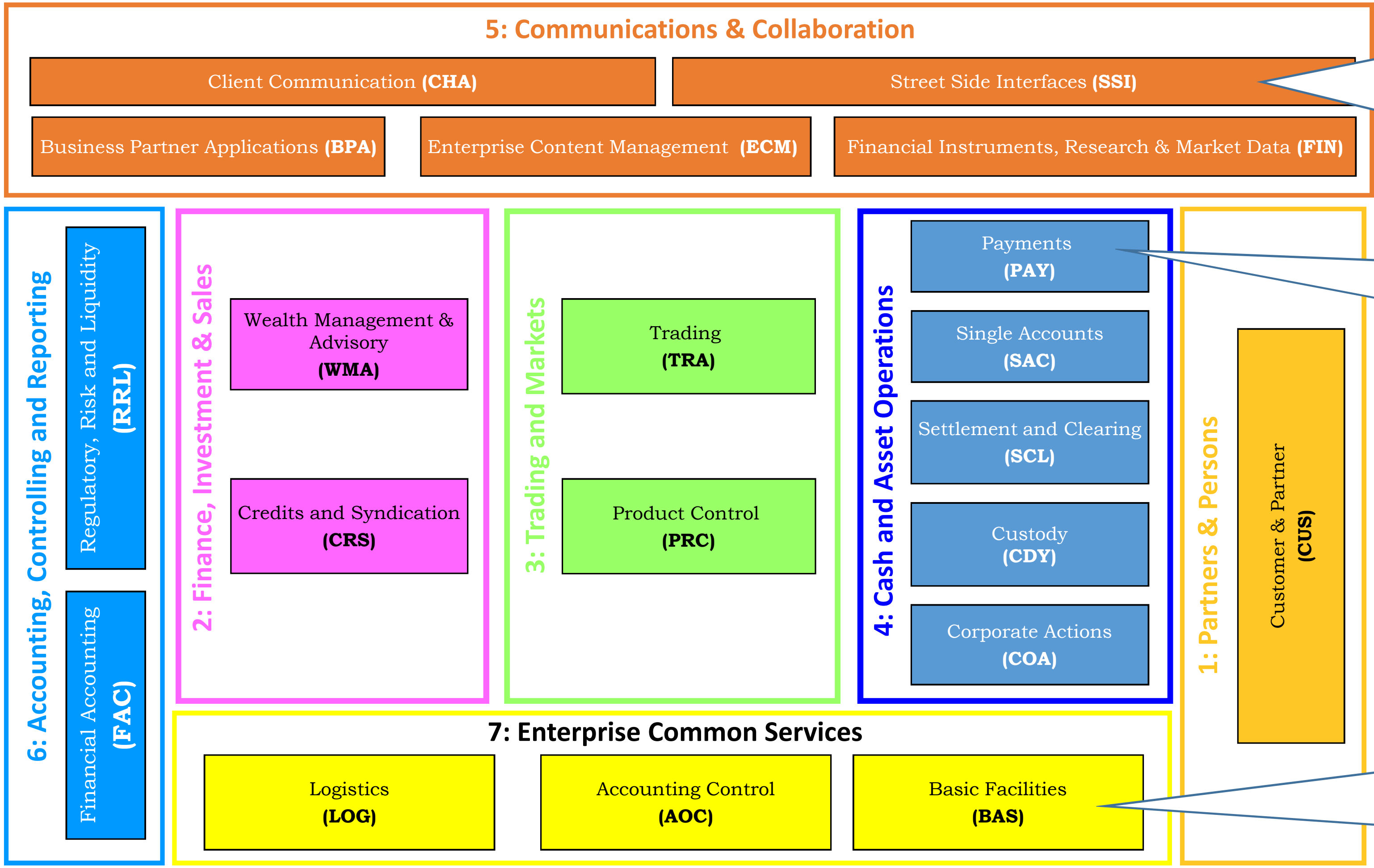
All domains = Full **compartimentalization** of the total bank functionality and data

Domain model:

Conceptual model defining the entities, their attributes, roles, relationships, and constraints that form an application domain

A domain model does not describe solutions to problems

Example: Domain Model for a Financial Institution



This domain contains all applications for the communication with exchanges, clearing etc.

This domain contains all applications for the payments

This domain contains all applications and information used in many other domains (Common functions)

Architecture

Business Object Model

Business Domain Objects

Wheel rotation sensor

Customer

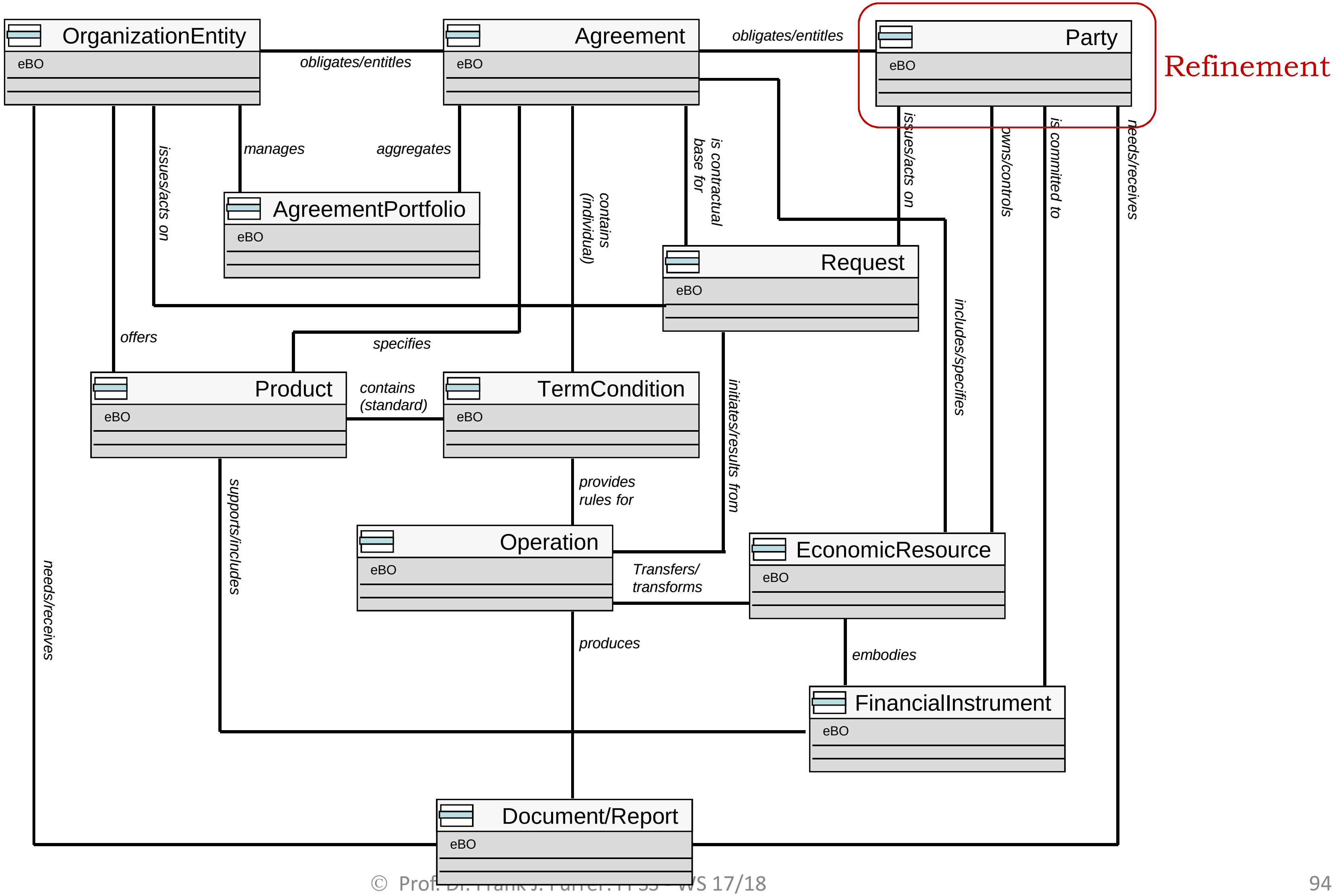
WheelRotationSensor
RotationRate
ReadRotationRate CalibrateZero

Properties → RotationRate

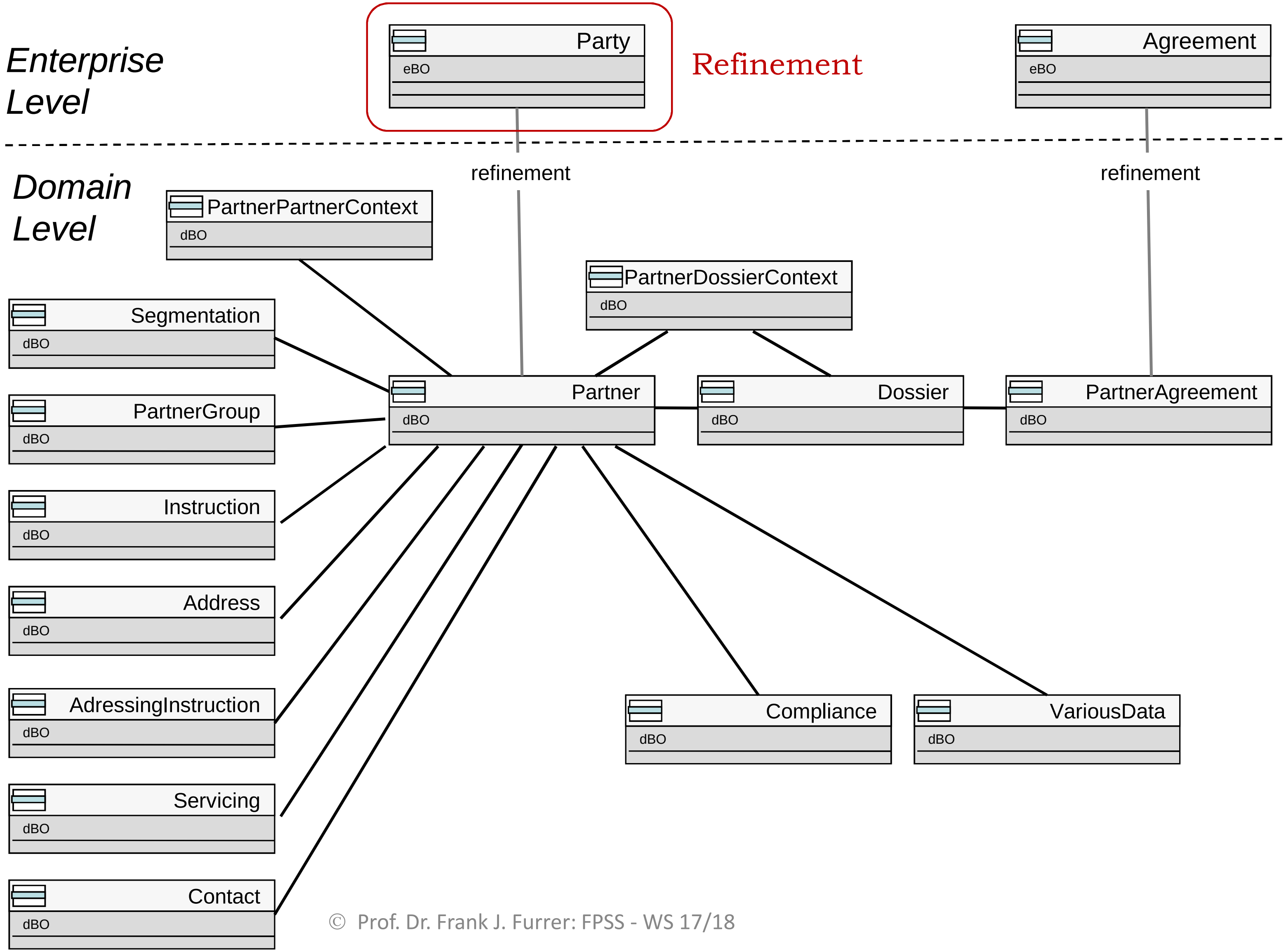
Methods → ReadRotationRate, CalibrateZero

Customer
Name Adress Nationality
Add/Delete/Archive Customer Read Properties Update Properties

Example: Financial Business Object Model (1/2)



Example: Financial Business Object Model (2/2)



A3

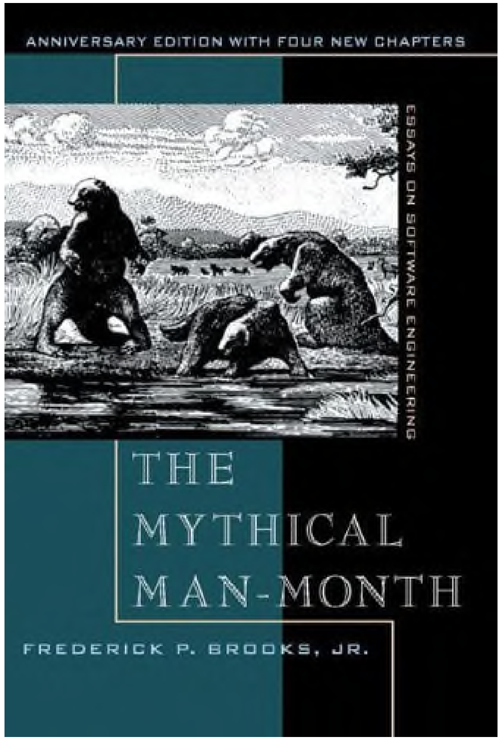
Architecture Principle A3:

Conceptual Integrity

1. Define all the concepts, the full terminology and models (including their relationships and relevant properties) precisely (whenever possible formally)
2. Draw the boundary of the system in which the definitions apply
3. Consistently and consequently use the definitions in all areas of the system
4. Strictly enforce the correct use of the definitions
5. When cooperating with systems outside the boundary, match the concepts and the terminology between all systems and interfaces

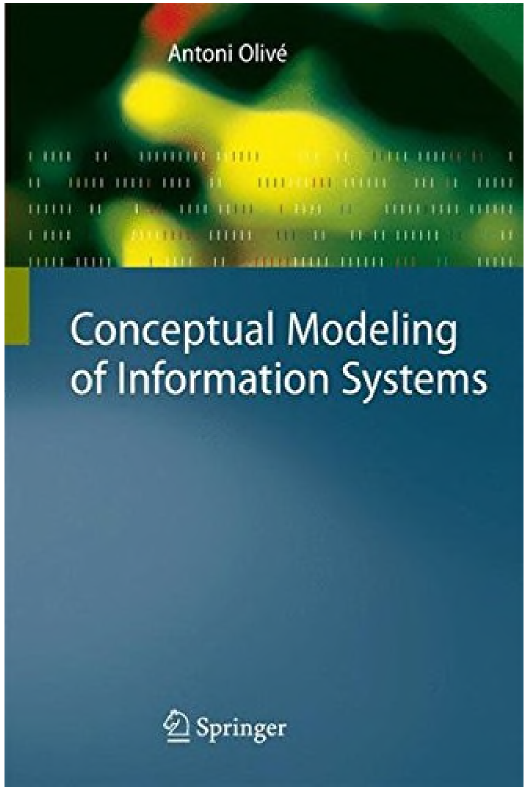
Justification: Misunderstandings between stakeholders lead to unsatisfactory IT-systems with divergence in many areas. Misunderstandings of all sorts must therefore be eliminated in all phases of systems engineering

Textbook



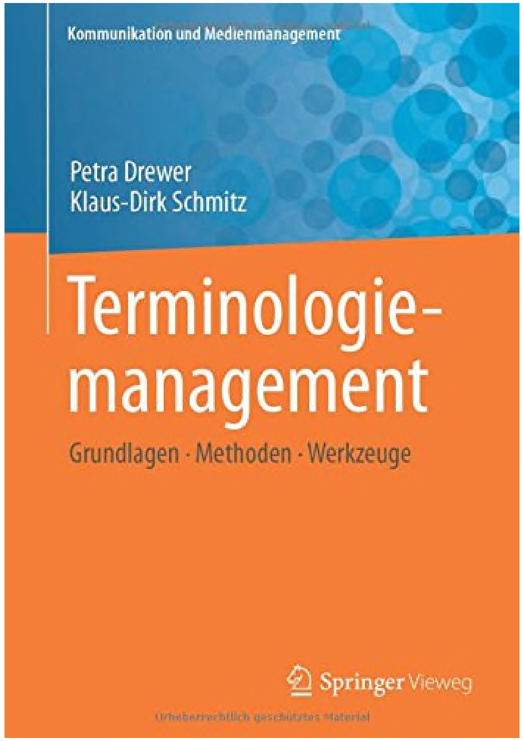
Frederick P. Brooks:
The Mythical Man-Month – *Essays on Software Engineering*
Addison-Wesley Longman (1975), New Edition, 1995. ISBN 978-0-201-83595-3

Textbook



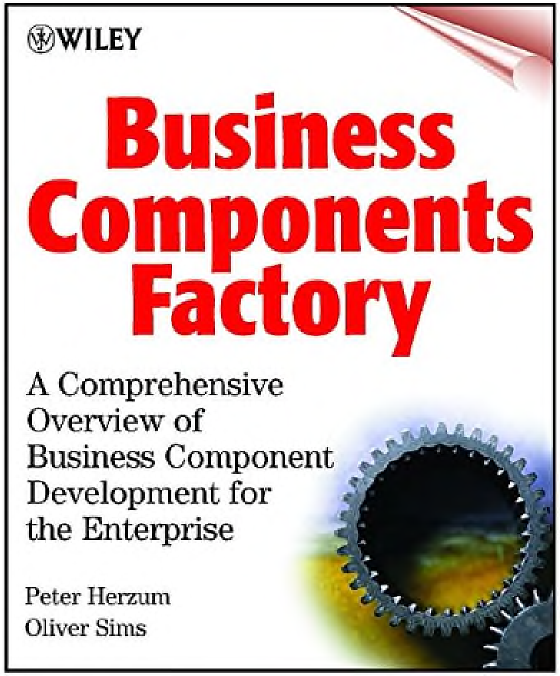
Antoni Olivé:
Conceptual Modeling of Information Systems
Springer-Verlag, Germany, 2007. ISBN 978-3-540-39389-4

Textbook



Petra Drewer, Klaus-Dirk Schmitz:
***Terminologiemanagement** – Grundlagen - Methoden – Werkzeuge*
Springer Vieweg Verlag, Germany, 2017. ISBN 978-3-6625-3314-7

Textbook



Peter Herzum, Oliver Sims:
***Business Component Factory** – A Comprehensive Overview of Component-Based Development for the Enterprise*
John Wiley & Sons Inc., USA, 2000. ISBN 978-0-471-32760-8

A4

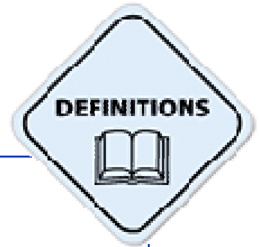
Architecture Principle A4:

Redundancy

Horizontal Architecture Layer Principles:

- A1: Architecture Layer Isolation
- A2: Partitioning, Encapsulation and Coupling
- A3: Conceptual Integrity
- A4: Redundancy
- A5: Interoperability
- A6: Common Functions
- A7: Reference Architectures, Frameworks and Patterns
- A8: Reuse and Parametrization
- A9: Industry Standards
- A10: Information Architecture
- A11: Formal Modeling
- A12: Complexity and Simplification

Redundancy in an IT-system is – in most cases – poison for the structure and for many quality properties of an IT-system



Definition:

Redundancy: The duplication of functionality or data as a whole or in parts

Redundancy Classification:

Requirements redundancy: The same or similar requirements are stipulated in different documents

Specification redundancy: Functional or data overlap in the specifications

Functional redundancy: The same or similar function is implemented several times in the IT-system

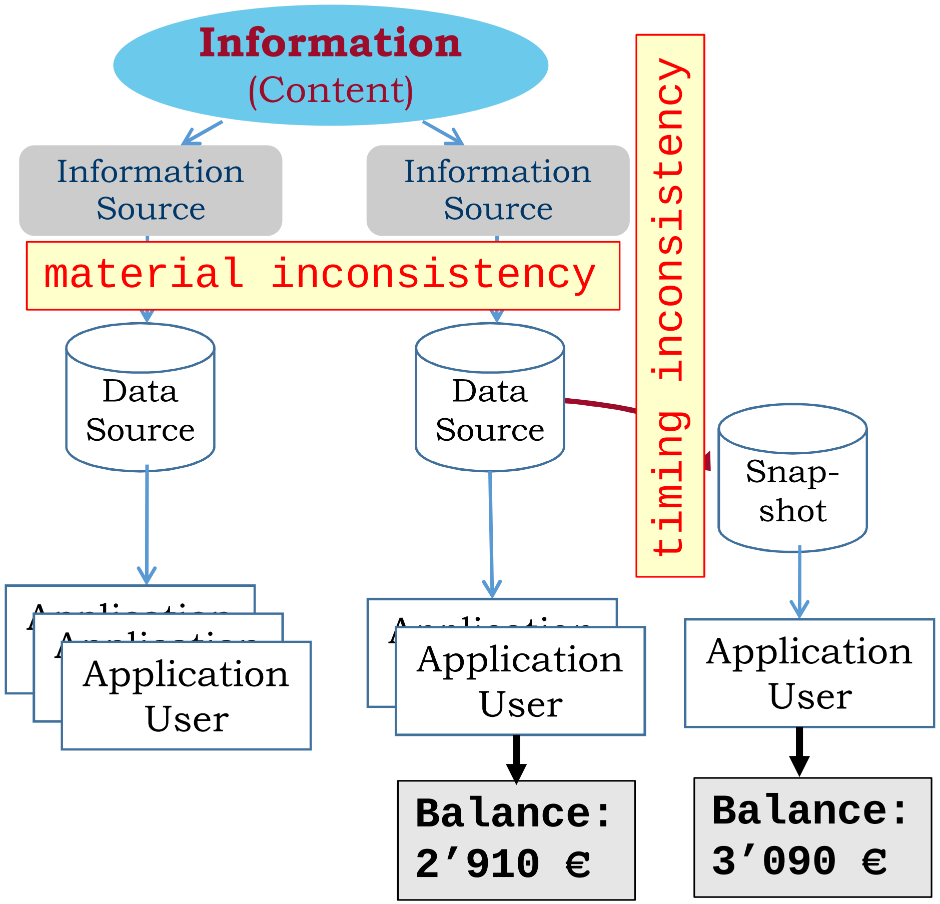
Data redundancy: Same elements of data are stored in different places and have different, unsynchronized sources

Interface redundancy: Interface functionality is implemented in more than one interface or overlaps interfaces

Code redundancy: The same or similar code-sequence is used in several programs

Implementation redundancy

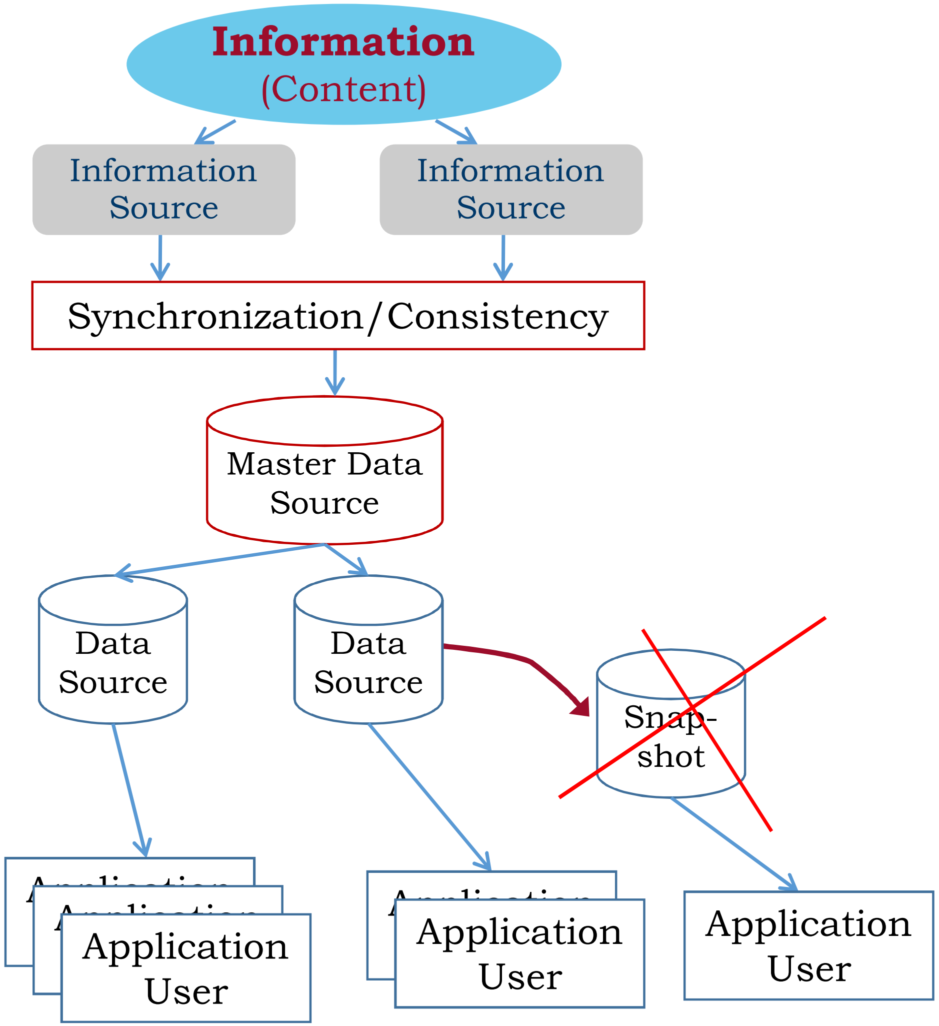
Example: Different applications work with inconsistent data (data redundancy)



Multiple, uncoordinated acquisition of the same information

Redundant, often inconsistent data

Applications or users work with different, inconsistent data



The data is generated, synchronized, and propagated in a consistent way

Applications or users work with consistent data

… on the other hand:
In some cases we ***need*** redundancy !

- disaster recovery (backup)
- high availability (faults & failures)
- load-balancing (multiple applications on multiple servers)
- performance requirements (parallel processing, DB accesses)
- geographical distribution (worldwide operations)
- electronic archiving
- 3^{rd} party software (sometimes difficult)
- safety (3-way voting)
- etc.

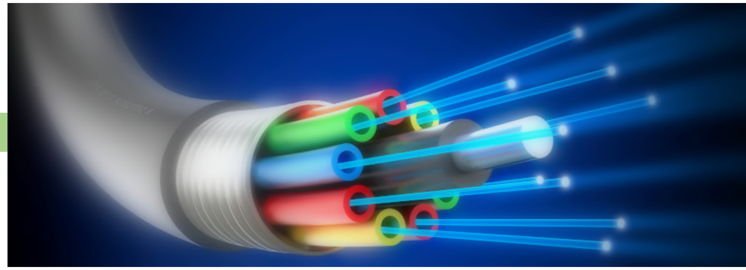
Wanted redundancy = *Managed* redundancy

Example: Transaction Data Base Mirroring

35 km

Main Computing System

Transaction Data Base

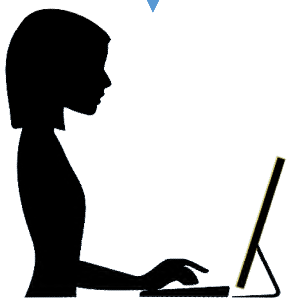


BackupComputing System

Backup
Transaction Data Base

Real-time copy
«Mirroring»

Interactive
User



= Managed Redundancy

… what is the solution?

Manage redundancy ! → **Managed redundancy**

	Managed redundancy	**Un**managed redundancy
Known and wanted	**Yes** (if valid reason)	**NO!**
Unknown or **un**wanted	**?**	**NO!**

DEFINITIONS

Managed redundancy definition:

- There is only exactly *one source* for the functionality and for the data (both during development time and during run-time)
- All redundant copies must be materially and time-wise synchronized (also partial copies)

Is the management of redundancy difficult?

http://www.shopclues.com/rubiks-cube-en-2.html

Yes!

... very

You need specific policies, processes and tools to successfully manage redundancy

⇒ and a strong awareness!

Redundancy is very difficult to identify and to eliminate – especially in large, complex IT systems

The redundancy-ghost

- **You don't hear it**
- **You don't see it**

Unmanaged redundancy ***infiltrates*** *the system via:*

- Requirements
- Specifications
- Architecting + Design Decisions
- Implementation (Evolution)
- Maintenance (adaptive and corrective)

http://gallery.yopriceville.com

The redundancy-ghost

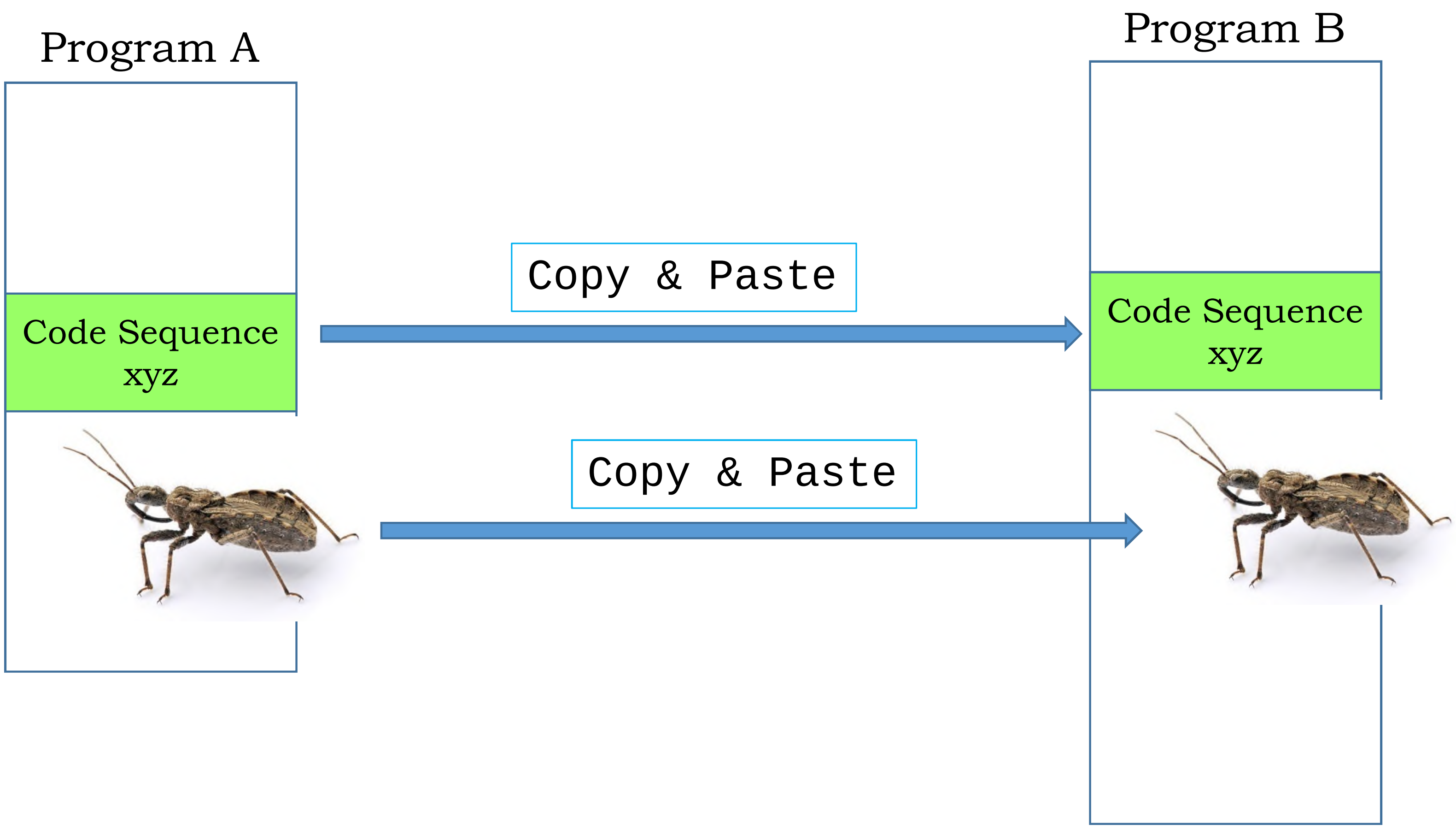
… but you see its impact !

During:

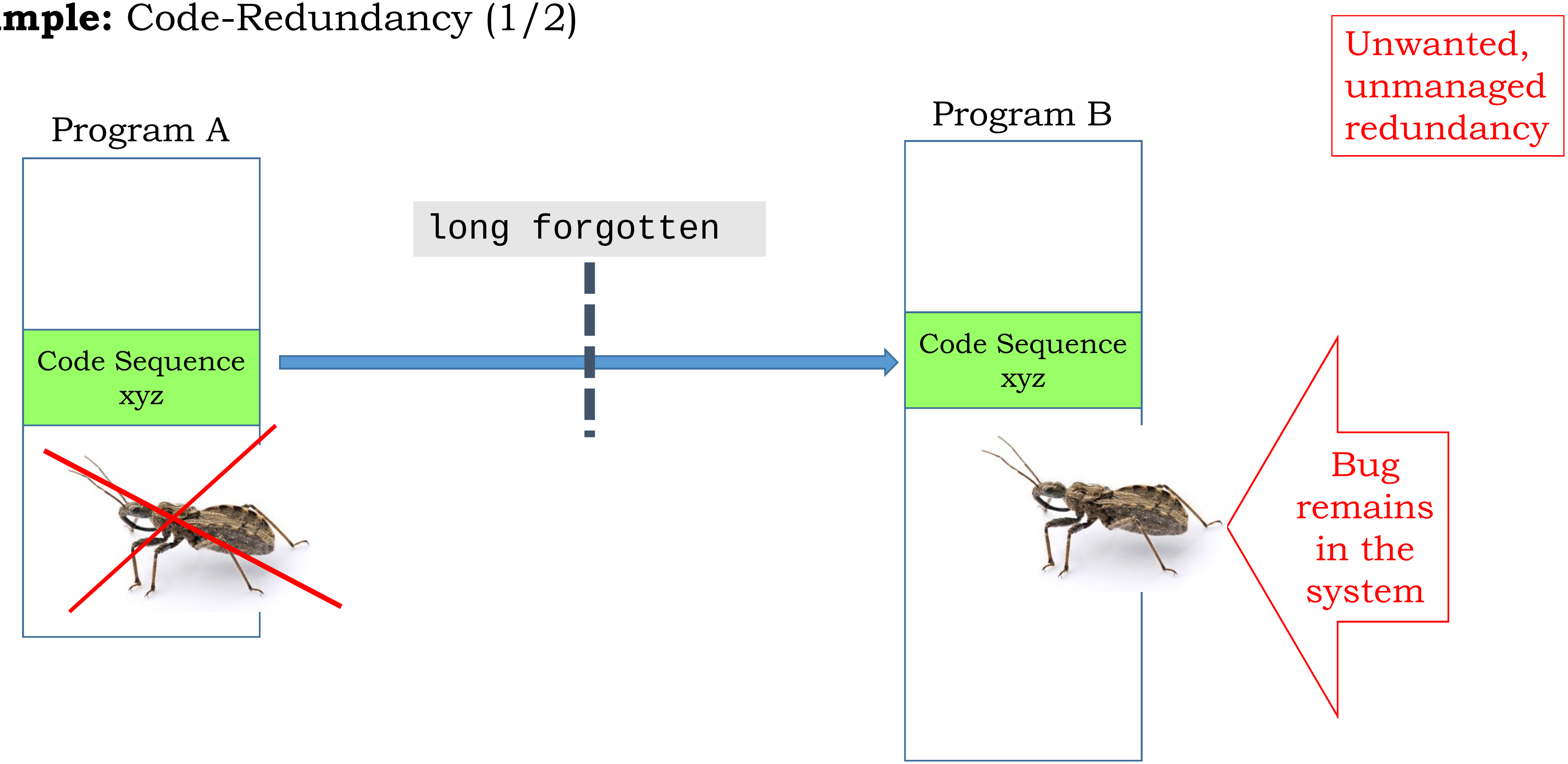
- **Operations**: inconsistent data & diverging functionality
- **Refactoring/Rearchitecting**: Hidden redundancy
- **Evolution** (Extensions): Changes in multiple parts
- **Maintenance** (Corrective): «search & hide»

Example: Code-Redundancy (1/2)

Unwanted, unmanaged redundancy

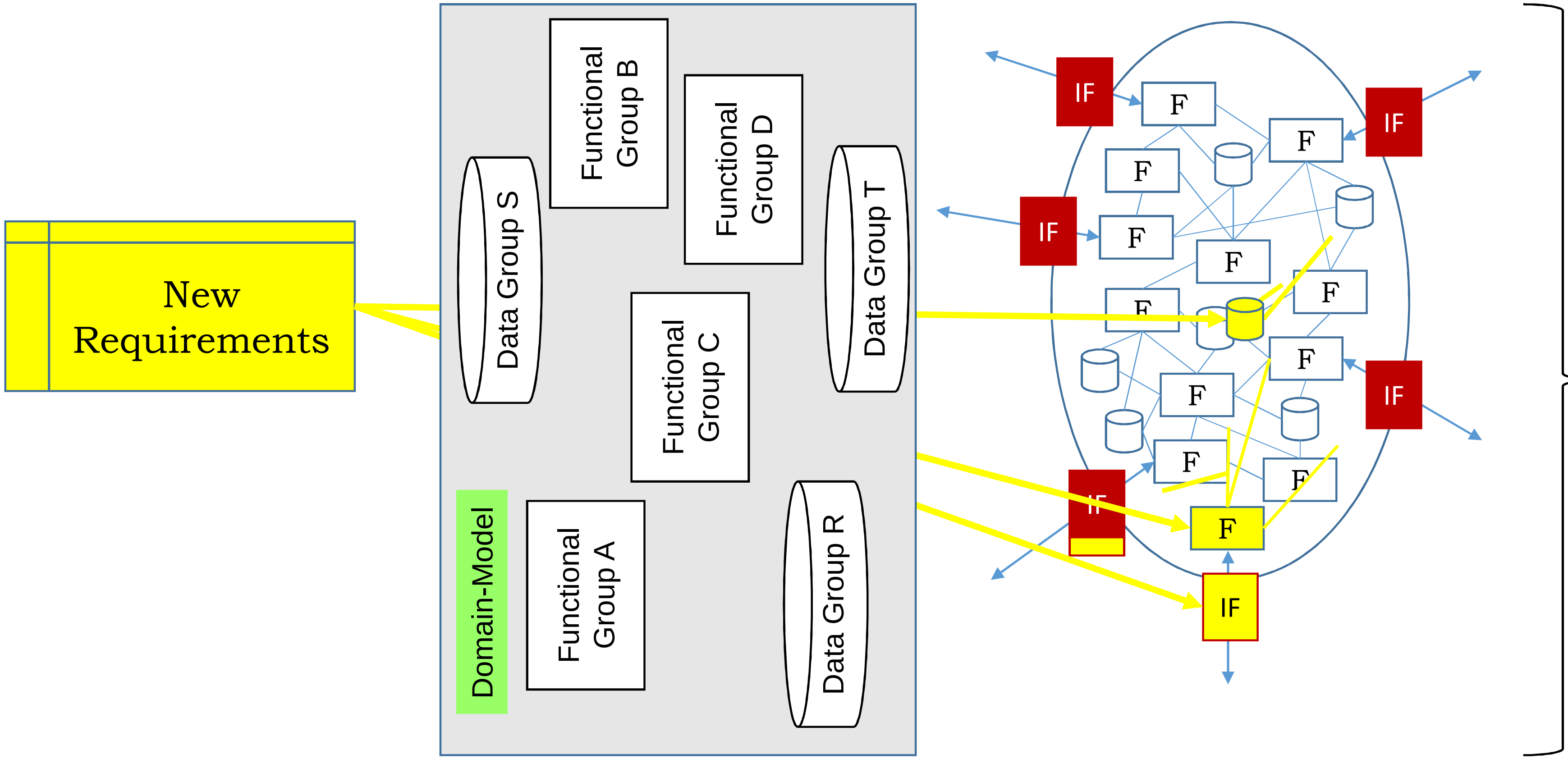


Example: Code-Redundancy (1/2)



Can we avoid the generation of unmanaged redundancy?

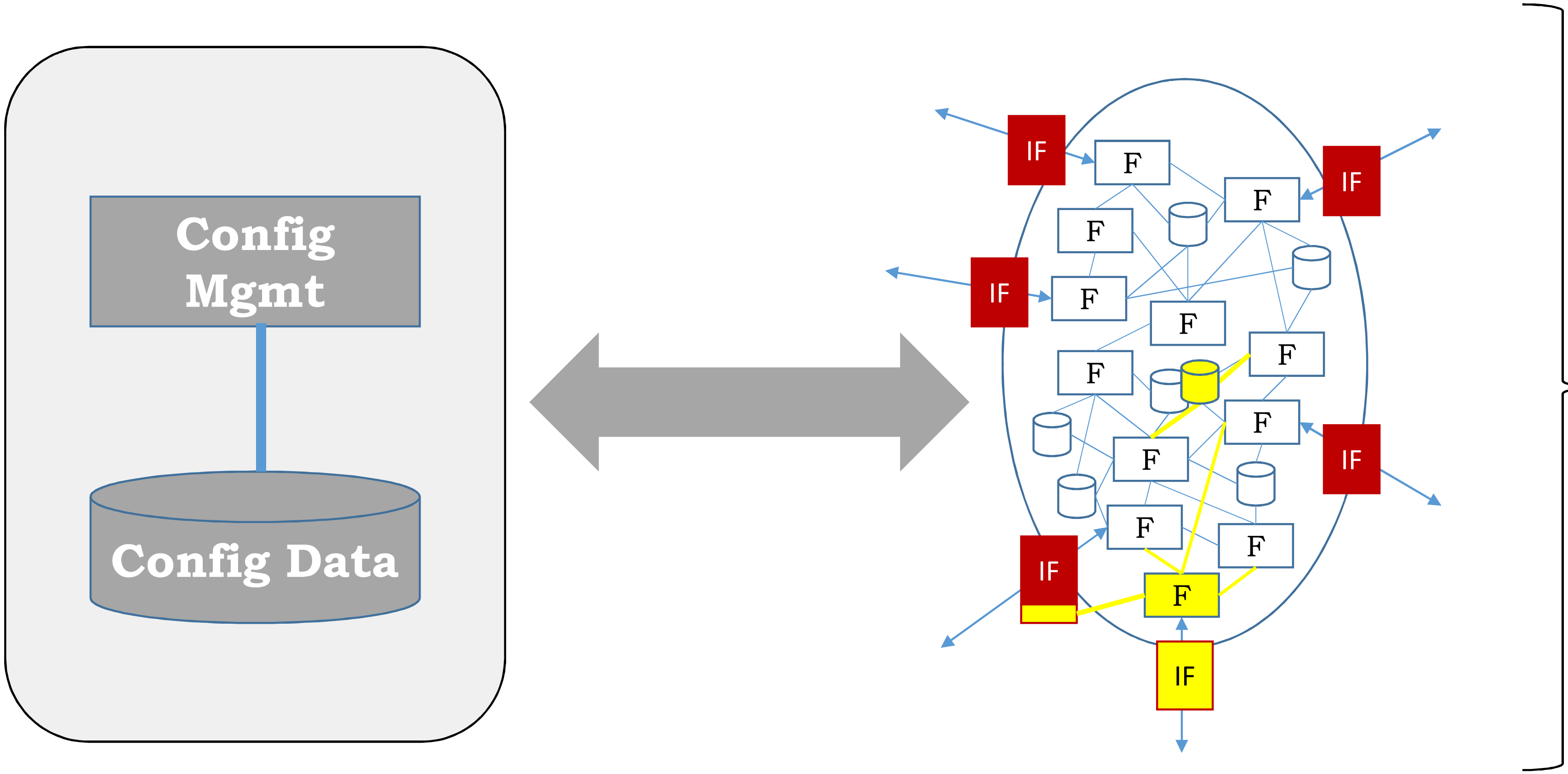
Tool 1: Domain-Model



The domain model *unambiguously* assigns data and functionality to one and only one part

Can we avoid the generation of unmanaged redundancy?

Tool 2: Configuration Management



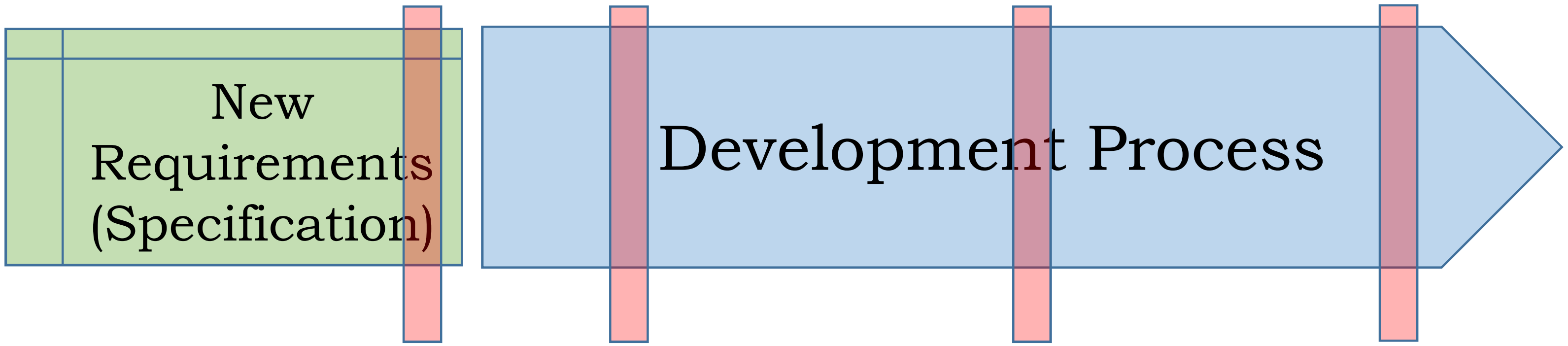
The configuration management data base records and manages all wanted redundancy
⇒ **Managed redundancy**

Can we avoid the generation of unmanaged redundancy?

Tool 3: Reviews

Redundancy ghost

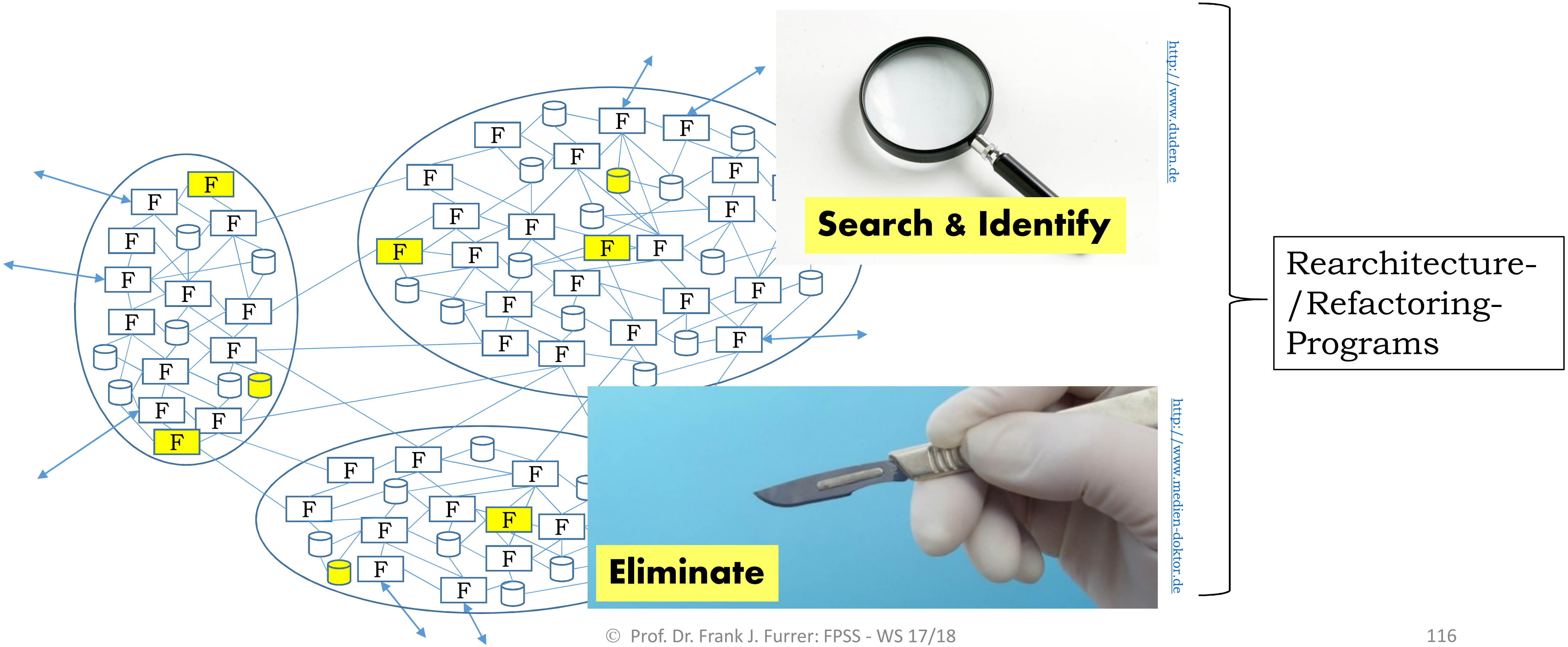
http://gallery.yopriceville.com



Check redundancy generation
(Reviews, Model-, Code-Checkers, …)

Existing **unmanaged** redundancy must be identified and eliminated

Tool 4: Rearchitecting/Refactoring



A4

Architecture Principle A4:

Redundancy

1. There is only exactly *one source* for the functionality and for the data (both during development time and during run-time)
2. All redundant copies must be content-wise and time-wise synchronized (thus avoiding divergence)
3. The creation of *unmanaged* redundancy is not allowed under any circumstances. Existing unmanaged redundancy must be identified and eliminated in due course
4. Managed redundancy is allowed if there is a good (documented) reason

Justification: Any unmanaged redundancy may cause divergence and thus severely impact quality properties of the system's output. Any unmanaged redundancy will negatively impact the maintenance and evolution of the system

A5

Architecture Principle A5:

Interoperability

Horizontal Architecture Layer Principles:

- A1: Architecture Layer Isolation
- A2: Partitioning, Encapsulation and Coupling
- A3: Conceptual Integrity
- A4: Redundancy
- A5: Interoperability
- A6: Common Functions
- A7: Reference Architectures, Frameworks and Patterns
- A8: Reuse and Parametrization
- A9: Industry Standards
- A10: Information Architecture
- A11: Formal Modeling
- A12: Complexity and Simplification

System Part **A** ⟷ System Part **B**

One of the most essential capabilities of collaborating systems is *interoperability*

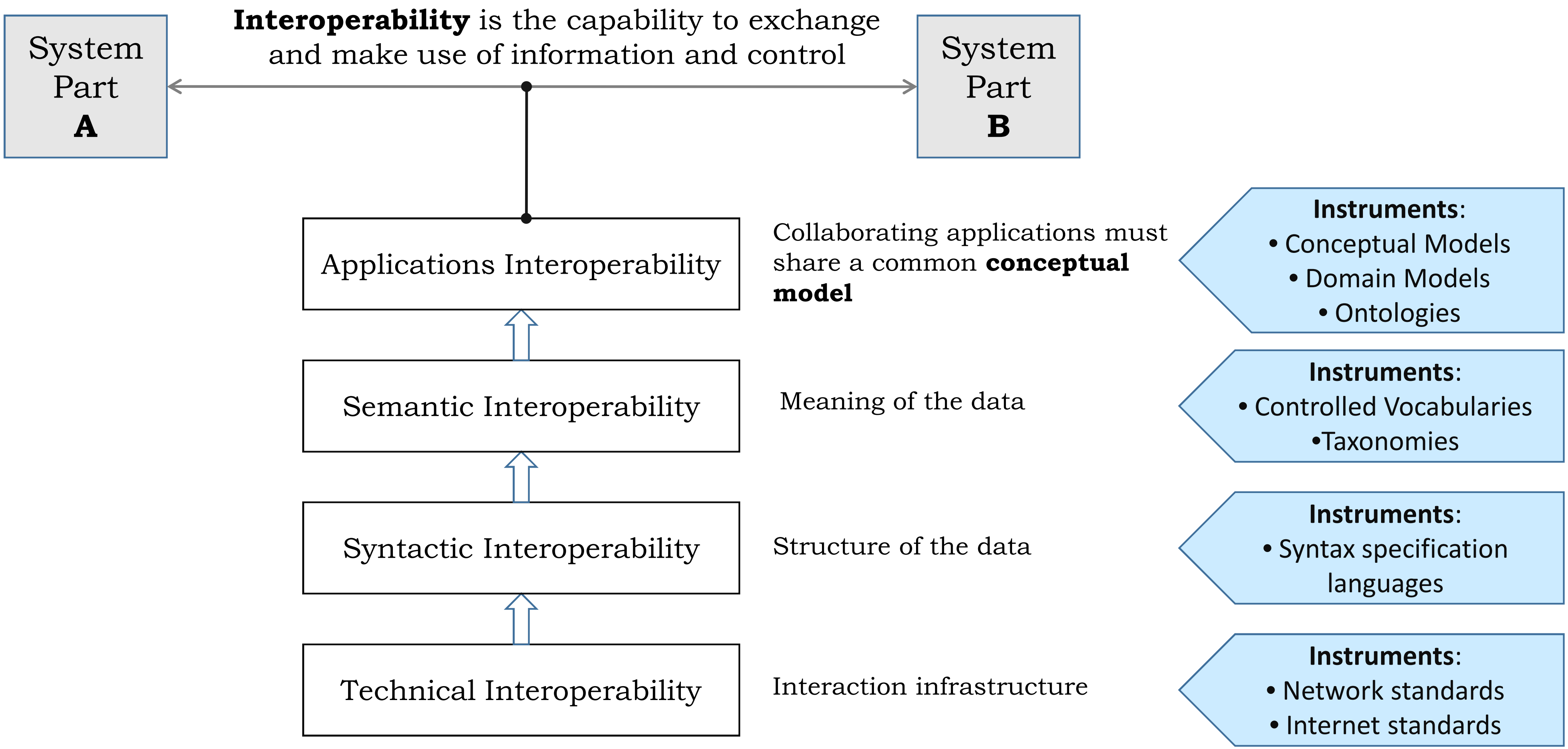
DEFINITIONS

Definition: **Interoperability** is the capability to exchange and make use of information and control

Interoperability must be assured on 4 levels:

- Applications Interoperability
- Semantic Interoperability
- Syntactic Interoperability
- Technical Interoperability

Interoperability Levels


System Part A
Interoperability is the capability to exchange and make use of information and control
System Part B
Applications Interoperability
Collaborating applications must share a common **conceptual model**
Instruments:
• Conceptual Models
• Domain Models
• Ontologies
Semantic Interoperability
Meaning of the data
Instruments:
• Controlled Vocabularies
•Taxonomies
Syntactic Interoperability
Structure of the data
Instruments:
• Syntax specification languages
Technical Interoperability
Interaction infrastructure
Instruments:
• Network standards
• Internet standards

Technical Interoperability

Example: Technical Error
(TSL Vulnerability)

In the SSL (Secure Socket Layer) and TLS (Transport Level Security)-protocol a **serious security vulnerability** was detected (November 2009)

RFC5746: SSL and TLS renegotiation are vulnerable to an attack in which the attacker forms a TLS connection with the target server, injects content of his choice, and then splices in a new TLS connection from a client.
http://tools.ietf.org/html/rfc5746

Because this security vulnerability is in the original ietf-*specification*, **all** SSL/TSL-implementations can be attacked via this vulnerability. All implementations and deployments must be patched – worldwide!

Syntactic Interoperability

http://www.izepcevski.com/mundane-explosions/

Example: Syntax Error
(Ariane 5 Explosion)

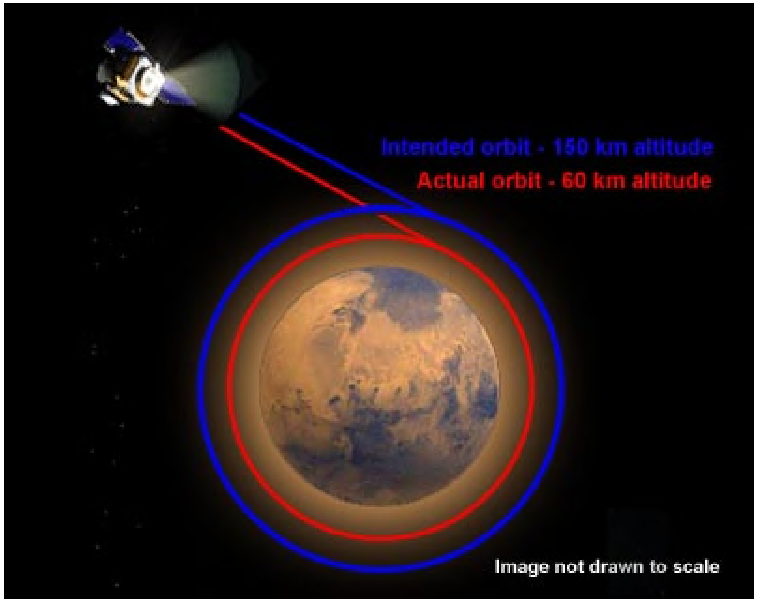
On 4th June 1996 the Ariane rocket exploded during its first commercial flight

The value for the **horizontal velocity** was stored as a *16-bit integer value* in the inertial guidance system, a heritage from the Ariane 4. The measured value, however, was stored in a *64-bit floating format*. Because the Ariane 5 was considerably faster than the Ariane 4, the conversion of the 64-bit floating value into the 16-bit integer value exceeded 32,767 and caused an operand error – resulting in the loss of guidance and the self-destruction of the rocket. This syntactic error caused losses of 1.7 billions of US$

(http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf).

Semantic Interoperability

http://www.vuw.ac.nz/staff/stephen_marshall/SE/Failures/media/MCO_Orbit.jpg



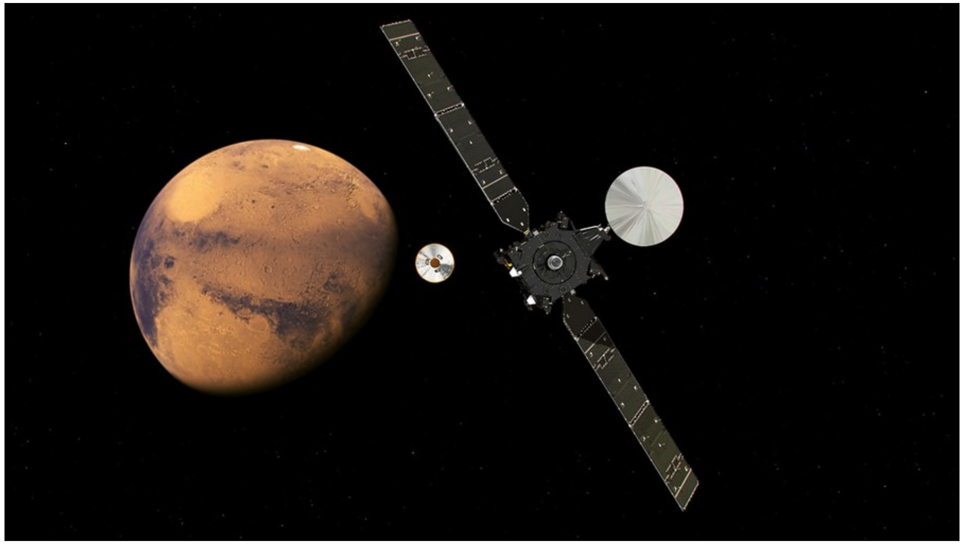
Example: Semantic Mismatch
(Mars Climate Orbiter Crash)

The Mars Climate Orbiter (MCO) mission objective was to orbit Mars as the first interplanetary weather satellite. The MCO was launched on December 11, 1998, and was lost sometime following the spacecraft's entry into Mars occultation during the Mars Orbit Insertion (MOI) maneuver.

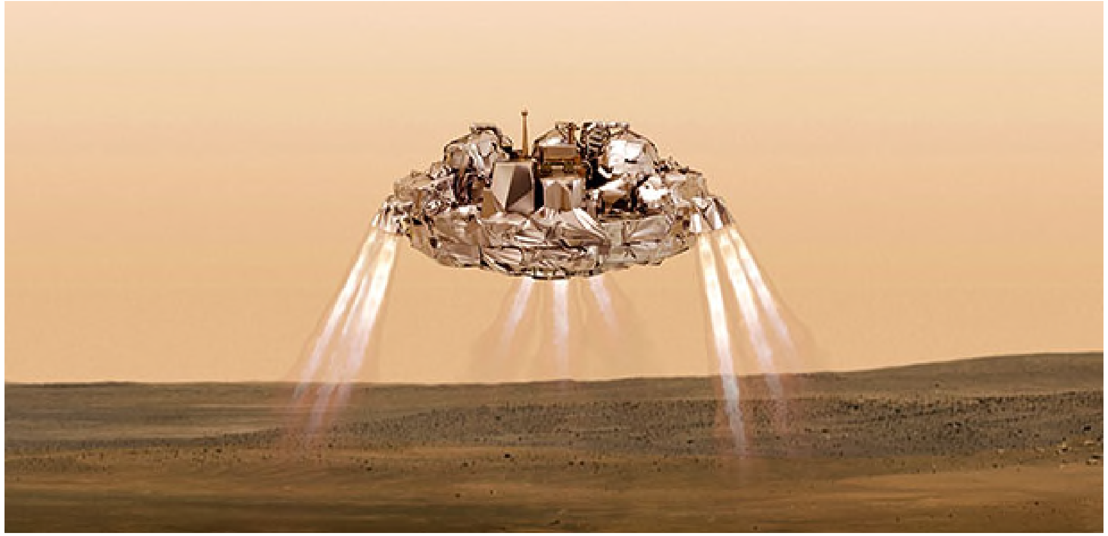
The root cause for the loss of the MCO spacecraft was a **semantic mismatch**. The **spacecraft software** was correctly programmed to *use metric units* (Newtonseconds). The **ground software** was programmed to use *English units* (pound-seconds). The same measurement values therefore had a different meaning – differing by a factor of 4.45 – in the spacecraft and in the ground software resulting in an erroneous trajectory and in the crash of the spacecraft. This semantic mismatch caused a loss of 193.1 million US$.
(ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf).

1996, 1998: Failed syntactic and semantic interoperability

Could that still happen in 2016?



http://www.aargauerzeitung.ch



http://www.scinexx.de

19. October <u>2016</u>:

The Mars Lander «**Schiaparelli**» crashes to the ground

First Analysis (November 2016)

https://de.wikipedia.org

Radar-Altimeter

http://www.militaryaerospace.com

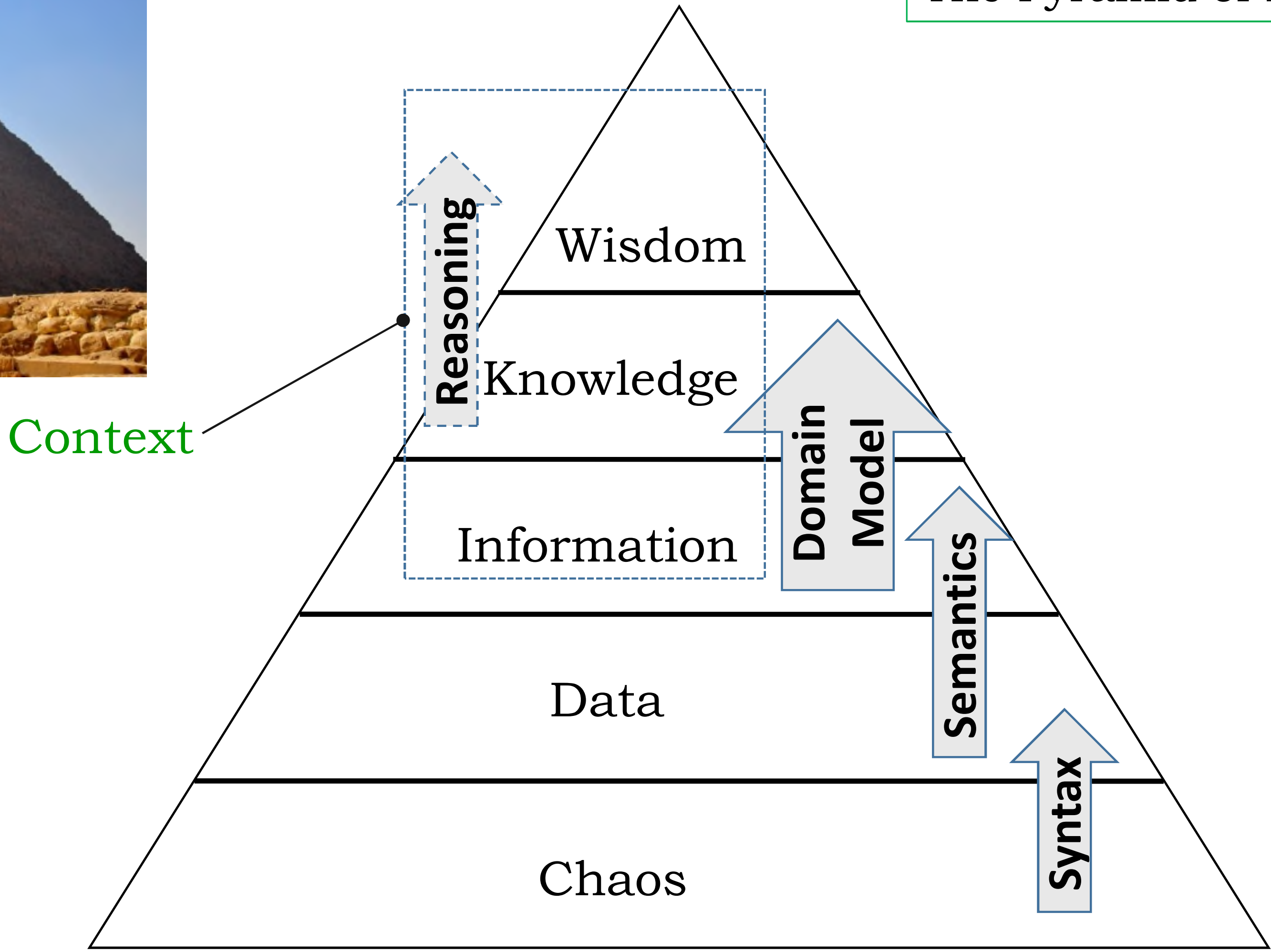
Navigation Computer

Software **Interoperability** problem between
Radar-Altimeter and Navigation Computer in the Lander

"A software error is ultimately good news for the ExoMars mission"
Andrea Accomazzo (ESA)

The Pyramid of Knowledge

http://ancienthistory.about.com/od/pyramids/tp/91012-The-Main-Pyramids-Of-Egypt.htm



The **context** of the specific information determines to a significant degree its full understanding and correct interpretation. Thus, for interoperability, the context of the semantic layer must be clearly defined

DEFINITIONS

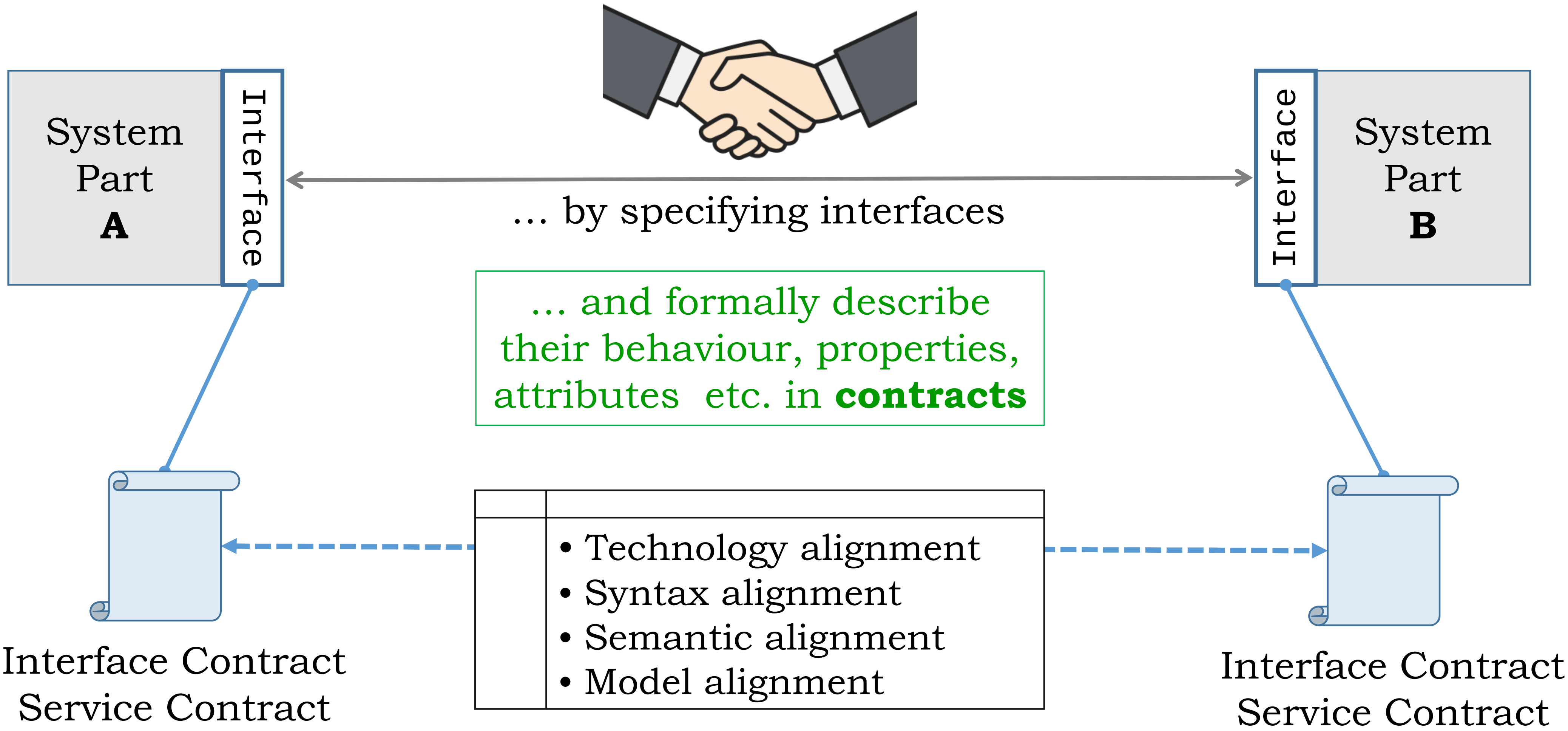
Definition: Context is the circumstances that form the setting for an event, statement, or idea and in terms of which it can be fully understood and assessed ([Oxford98]).

Example: Context Mismatch
The American and the Russian president agree on a running competition over 5 km. The American president clearly wins the race. The reporting in the Russian press reads: "***The Russian president ran to an excellent second place, whereas the American president only finished second last in the race***".

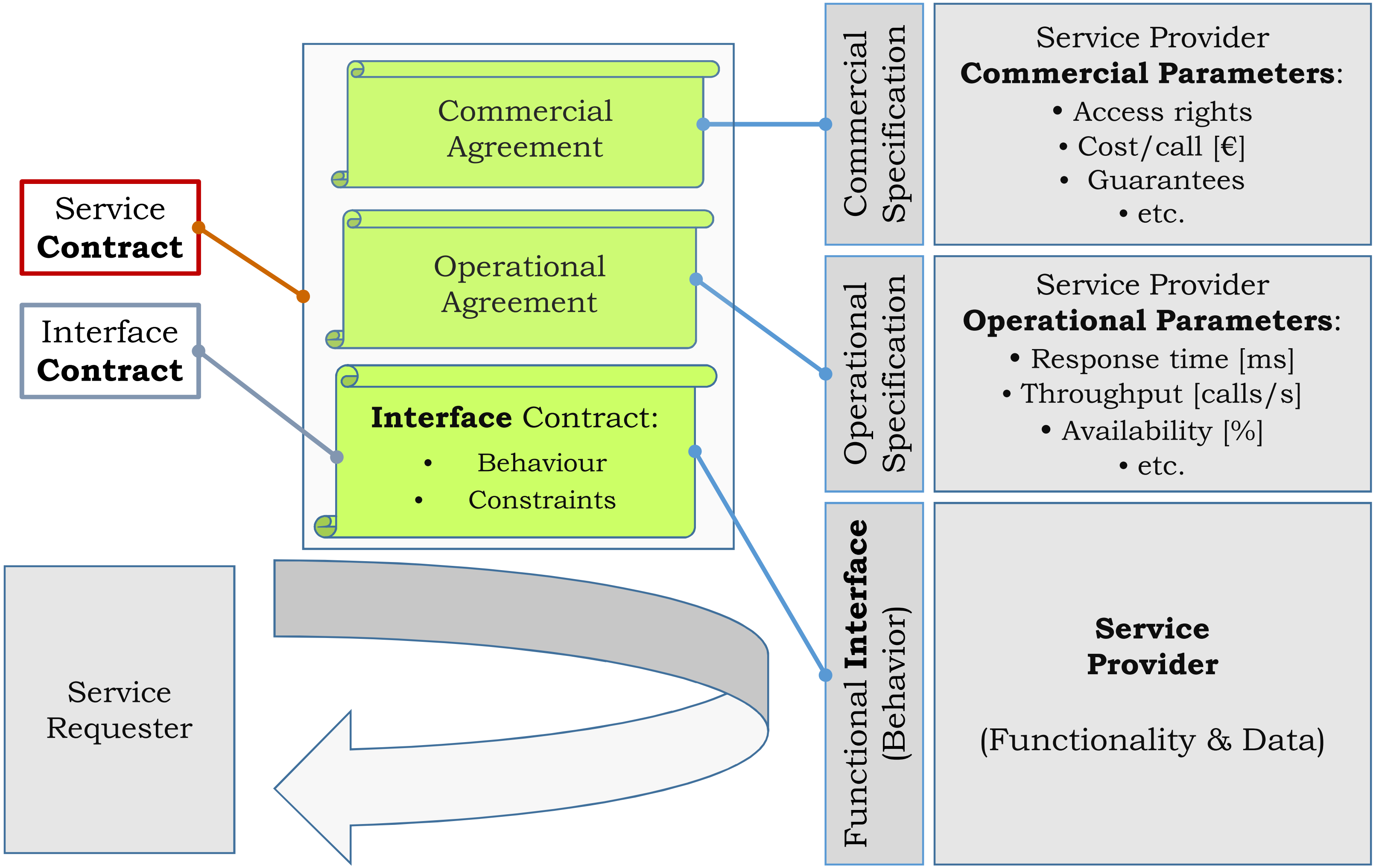
Without knowing the *context* that only 2 competitors ran the race, the meaning of the information is completely distorted – people implicitly assume that the race had 50 … 100 runners.

http://www.link-gr.ch/news/20080628/graubuenden-marathon-wyatt/

How can we assure interoperability?



Interface & Service **Contracts**:



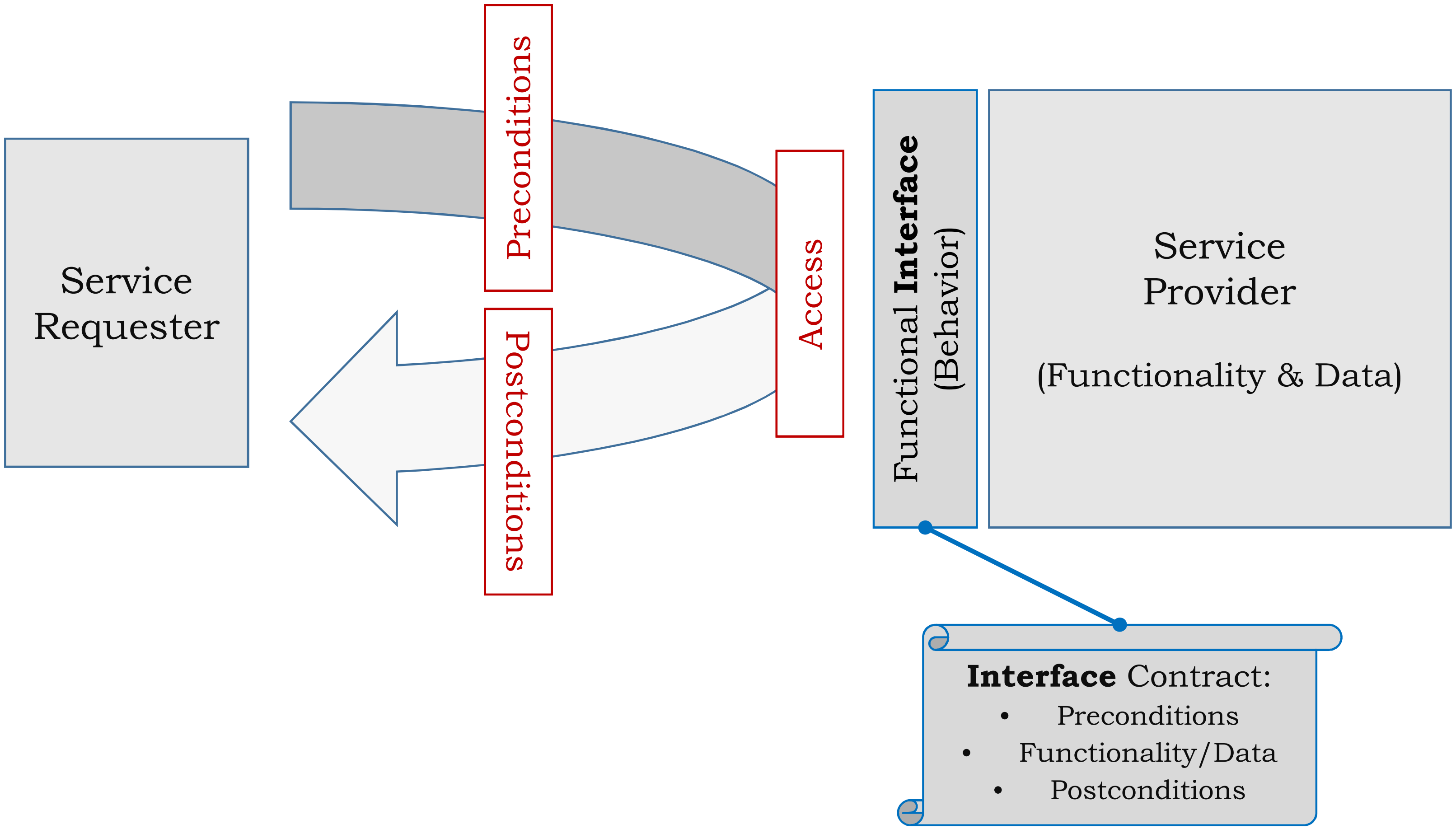
Example:
Simple Web-Service Formalization

Web service which works as a *service provider* and exposes two methods (*add and SayHello*) as the web services to be used by applications

```
<%@ WebService language = "C#" class = "FirstService" %>

using System;
using System.Web.Services;
using System.Xml.Serialization;

[WebService(Namespace="http://localhost/MyWebServices/")]
public class FirstService : WebService{
  [WebMethod]
  public int Add(int a, int b) {
    return a + b;
  }

  [WebMethod]
  public String SayHello() {
    return "Hello World";
  }
}
```

https://www.tutorialspoint.com/webservices/web_services_examples.htm

Interface Contracts:



Example:
Add new customer (ID and name)

Precondition:
1. ID not already active

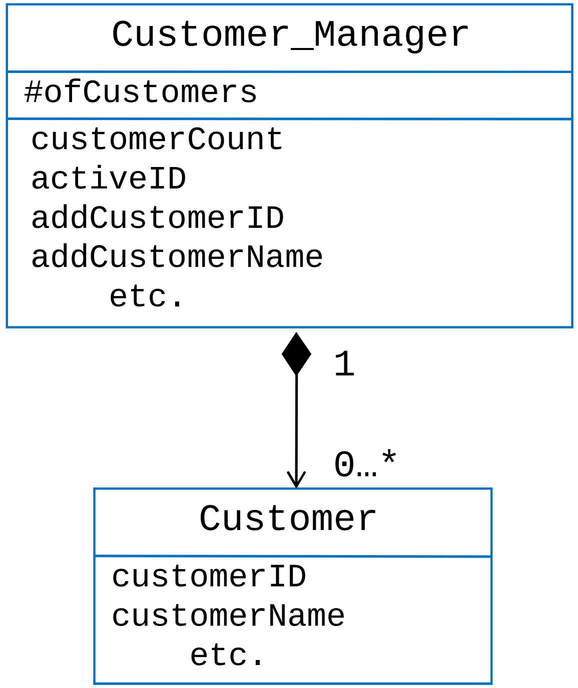
addCustomerID

Postconditions:
1. ID now active
2. #ofCustomers + 1

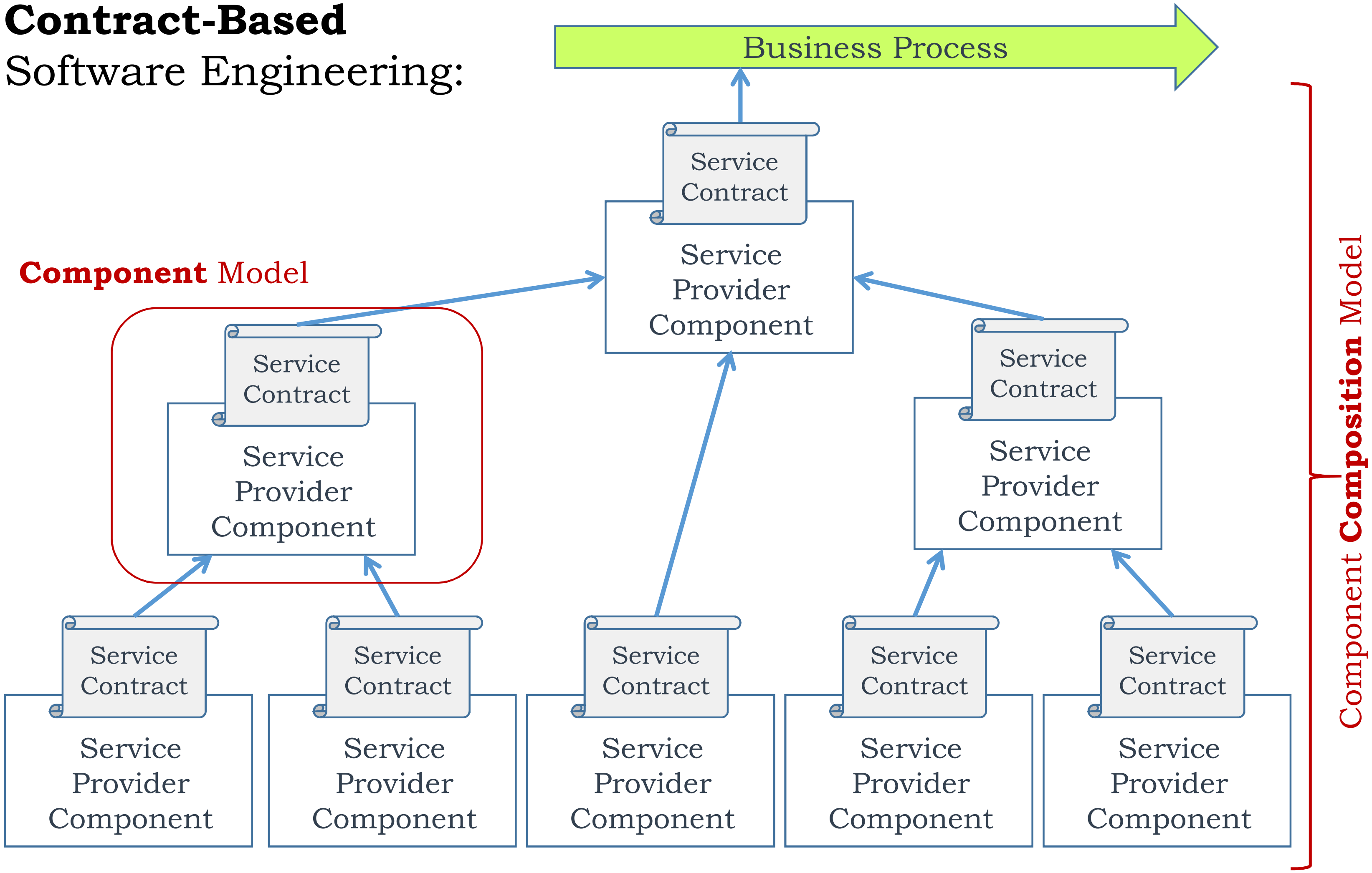
Precondition:
1. ID active

addCustomerName

Postcondition:
1. Name registered



Contract-Based
Software Engineering:



The *functionality* of a system is generated by ***composing*** service contracts

Service-Oriented Architecture (SOA)

see other TU-Dresden Lectures

http://integrella.com/services/solutions/a2a-and-b2b-integration/

System Part **A**

Interoperability is the capability to exchange and make use of information and control

System Part **B**

Applications Interoperability — Collaborating applications must share a common **conceptual model**

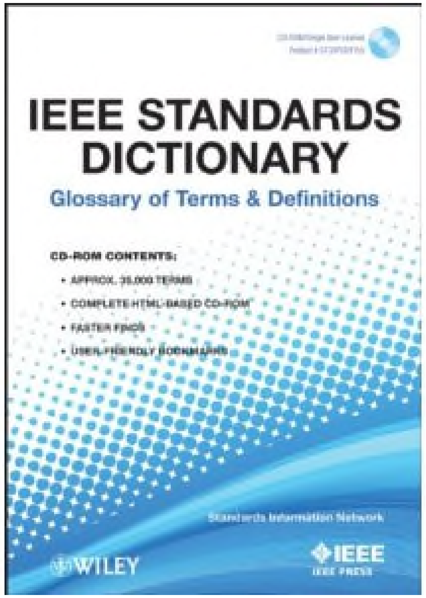
Semantic Interoperability — Meaning of the data

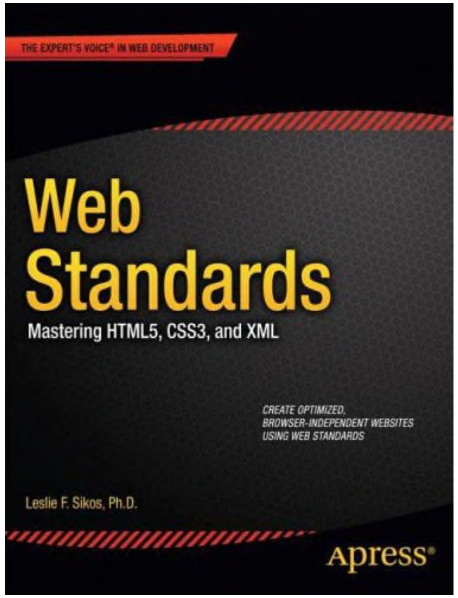
Syntactic Interoperability — Structure of the data

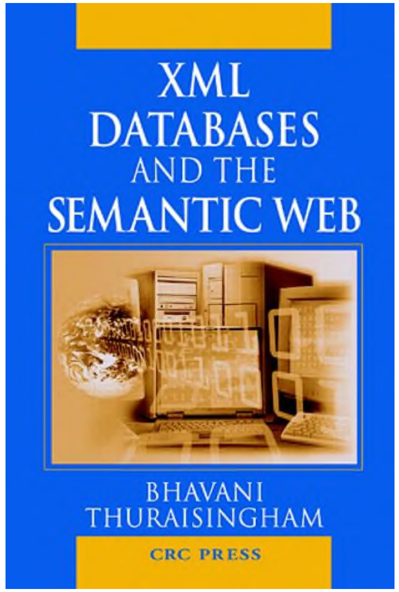
Technical Interoperability — Interaction infrastructure

How do we attain that?

How do we attain **Technological Interoperability**?







… by adapting and enforcing accepted industry standards

How do we express **Syntax**? ← machine-readable!

Def: The Extensible Markup Language (**XML**) is a markup language defining a set of rules for representing and encoding „documents“ in machine-readable formats

Note 1: XML is in fact a technology to generate specific markup languages, e.g. domain-languages

Note 2: Today hundreds of specific XML-formats for different purposes exist

Note 3: XML is also human-readable (once you get used to the format)

Tags defined by the author of this document

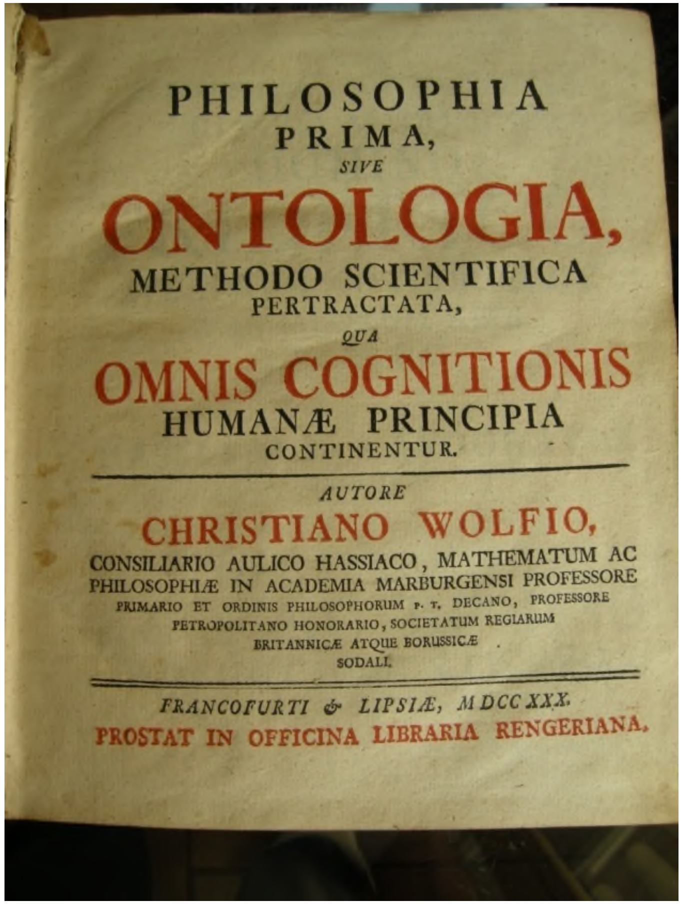
```
<?xml version="1.0"?>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

[http://www.w3schools.com/xml/]

Example:
XML Service Syntax

```
<!--
######################################################################################
                    Service-Id:     DOC_1042
                    Service-Name:   Request Digital Signature
                    Technical Name:   Request Digital Signature for an electronic document or a data structure
                    History:           v0.1   03.06.2005    Draft
                                       v1.0   01.07.2005    Final
######################################################################################
-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
       xmlns:cs="http://www.cs-standards.org/schema/CS-BASE-1-0"
                              xmlns:cif="http://www.cs-standards.org/schema/CS-CIF-BASE-1-0"
                              xmlns:ebi="http://www.cs-standards.org/schema/CS-EBI-BASE-1-0"
                              xmlns:dss="http://www.cs-standards.org/schema/CS-DSS-BASE-1-0"
                              elementFormDefault="unqualified" attributeFormDefault="qualified">

          <xs:import namespace="http://www.cs-standards.org/schema/CS-BASE-1-0"     schemaLocation="CS-BASE-1-0.xsd"/>
          <xs:import namespace="http://www.cs-standards.org/schema/CS-CIF-BASE-1-0" schemaLocation="CS-CIF-BASE-1-0.xsd"/>
          <xs:import namespace="http://www.cs-standards.org/schema/CS-EBI-BASE-1-0" schemaLocation="CS-EBI-BASE-1-0.xsd"/>
          <xs:import namespace="http://www.cs-standards.org/schema/CS-DSS-BASE-1-0" schemaLocation="CS-DSS-BASE-1-0.xsd"/>

          <!--
          ==================================================================================
             ELAR Signature Request slot: 16 ELAR signature requests can be grouped into a single message.
          ==================================================================================
          -->
          <xs:complexType name="RequestSlotType">
                    <xs:sequence>
                              <xs:element name="SlotStatus"    type="dss:SlotStatus"/>
                              <xs:element name="BusinessUnit"   type="cs:businessUnitType"/>
                              <xs:element name="HashtreeUUID"   type="cs:uuid20AsCharType"/>
                              <!--
                              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

How do we express **Semantics**? ← `machine-readable!`



PHILOSOPHIA
PRIMA,
SIVE
ONTOLOGIA,
METHODO SCIENTIFICA
PERTRACTATA,
QUA
OMNIS COGNITIONIS
HUMANÆ PRINCIPIA
CONTINENTUR.
AUTORE
CHRISTIANO WOLFIO,
CONSILIARIO AULICO HASSIACO, MATHEMATUM AC
PHILOSOPHIÆ IN ACADEMIA MARBURGENSI PROFESSORE
PRIMARIO ET ORDINIS PHILOSOPHORUM P. T. DECANO, PROFESSORE
PETROPOLITANO HONORARIO, SOCIETATUM REGIARUM
BRITANNICÆ ATQUE BORUSSICÆ
SODALI.
FRANCOFURTI & LIPSIÆ, MDCCXXX.
PROSTAT IN OFFICINA LIBRARIA RENGERIANA.

“Christiano Wolfio: Ontologia“, 1730

→ Ontology

Information Technology:

Def: An **ontology** is a formal representation of the *knowledge in a domain*

in the form of the *concepts* of the domain and their *relationships*,

and the *properties* of the concepts and relationships,

as well as the axioms and *principles* which are valid in the domain.

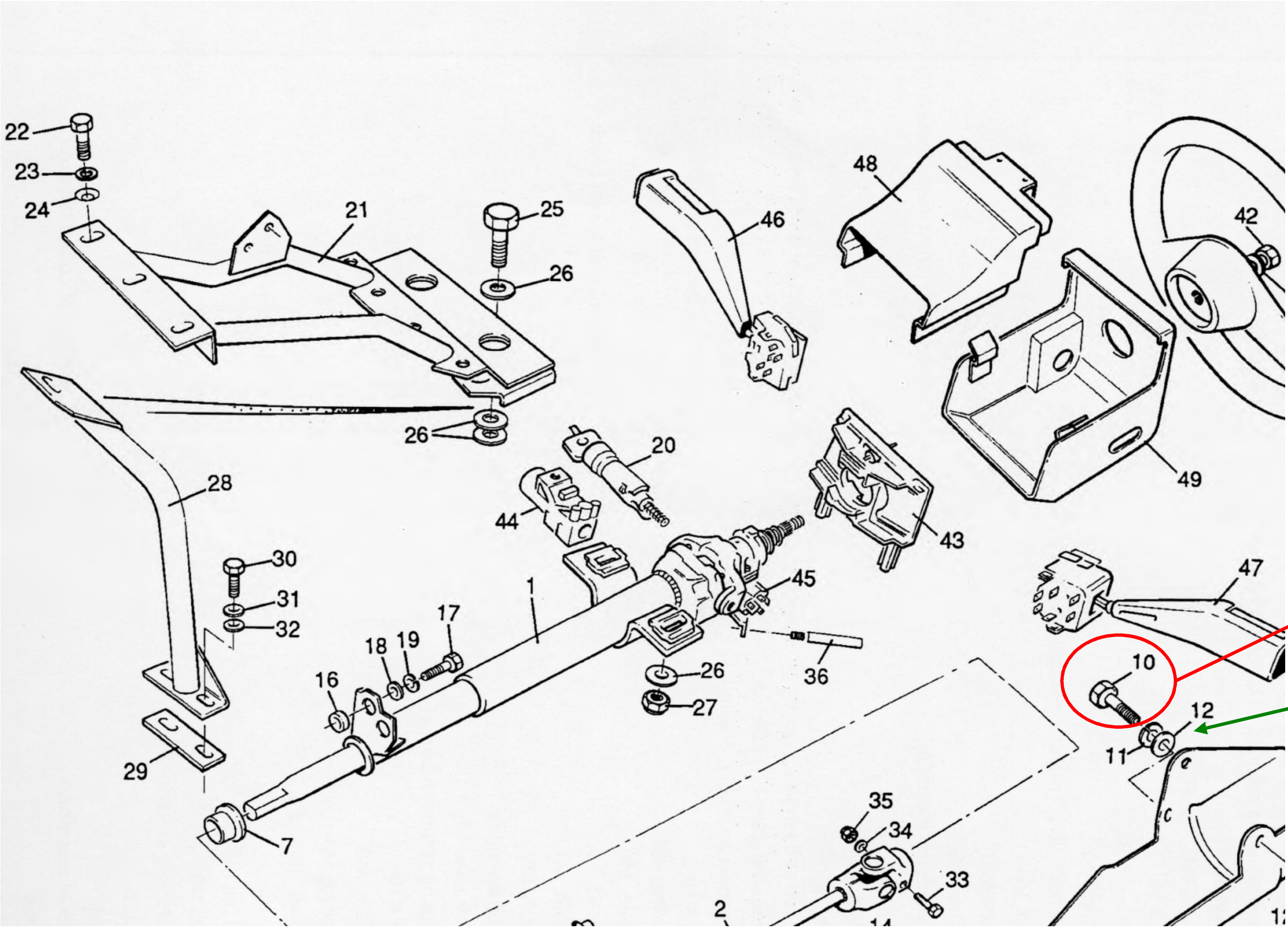
DEFINITIONS

Example: Car Ontology (1/5)

Aston Martin Virage 1991

Example: Car Ontology (2/5)

Aston Martin Virage Shop Manual



The **Car Shop Manual** contains all the parts and relationship information
(in graphical form)

Example: Car Ontology (3/5)

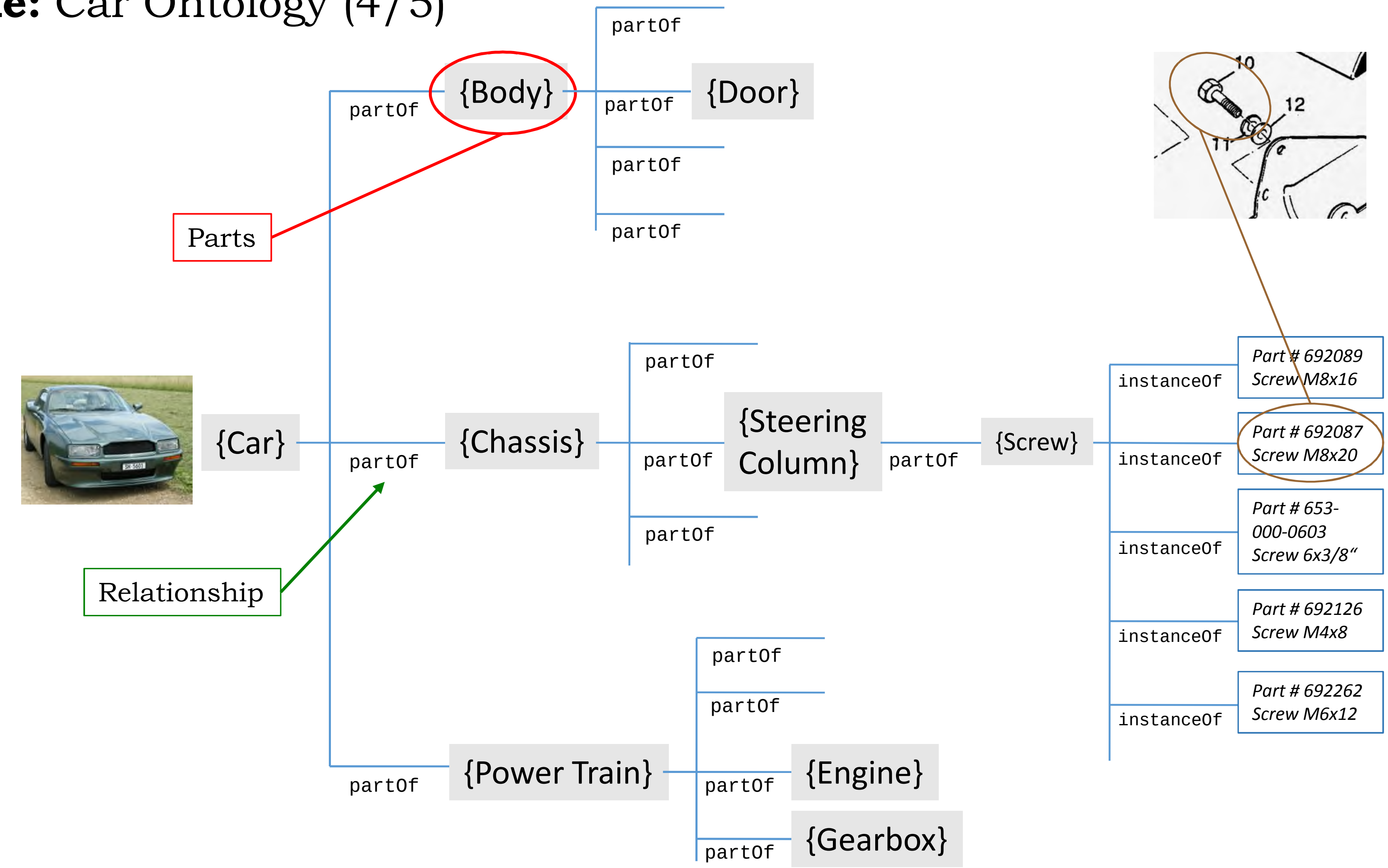
Virage Volante

4.3A Suspension and Steering
Steering Column

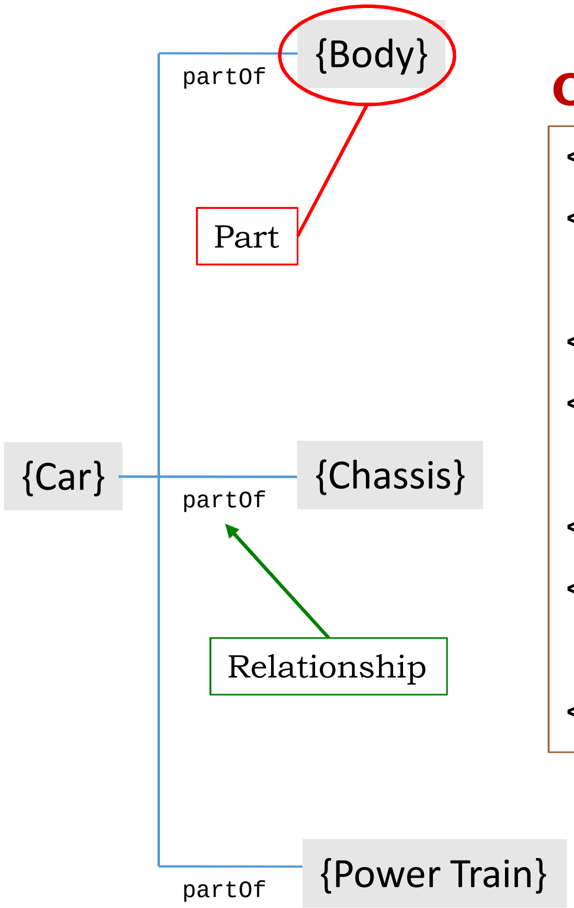
Item	Pt Number	Description	Qty	Remarks
1	25-20371	Steering column	1	
2	25-55229	Steering column lower assembly	1	
3	25-51824	Gaiter, steering column	1	
4	25-51828	Bush, steering gaiter	2	
5	25-51831	Spring, steering gaiter	2	
6	25-52117	Mounting bracket, lower column	1	
7	25-20191	Bearing, lower steering column. I.D.25.45/25.50mm	1	
-	25-21125	Bearing, lower steering column. I.D.25.50/25.53mm	1	Code: Red Alternatives
-	25-21126	Bearing, lower steering column. I.D.25.55/25.58mm	1	Code: Green " "
-	25-21127	Bearing, lower steering column. I.D.25.60/25.63mm	1	Code: Blue " "
8	25-52122	Angle bracket, RH	1	
9	25-52123	Angle bracket, LH	1	
10	692088	Screw, M6 x 16	4	Angle brkts & lwr shroud to mtg brkt
11	692046	Washer, spring, M6	4	" " " "
11A	692056	Washer, plain, M6	4	" " " "
12	25-54458	Washer, special. (RHD manual gearbox cars only)	3	" " " "
13	692089	Screw, M8 x 16	4	Angle bracket to crossmember

Properties

Example: Car Ontology (4/5)



Example: Car Ontology (5/5)

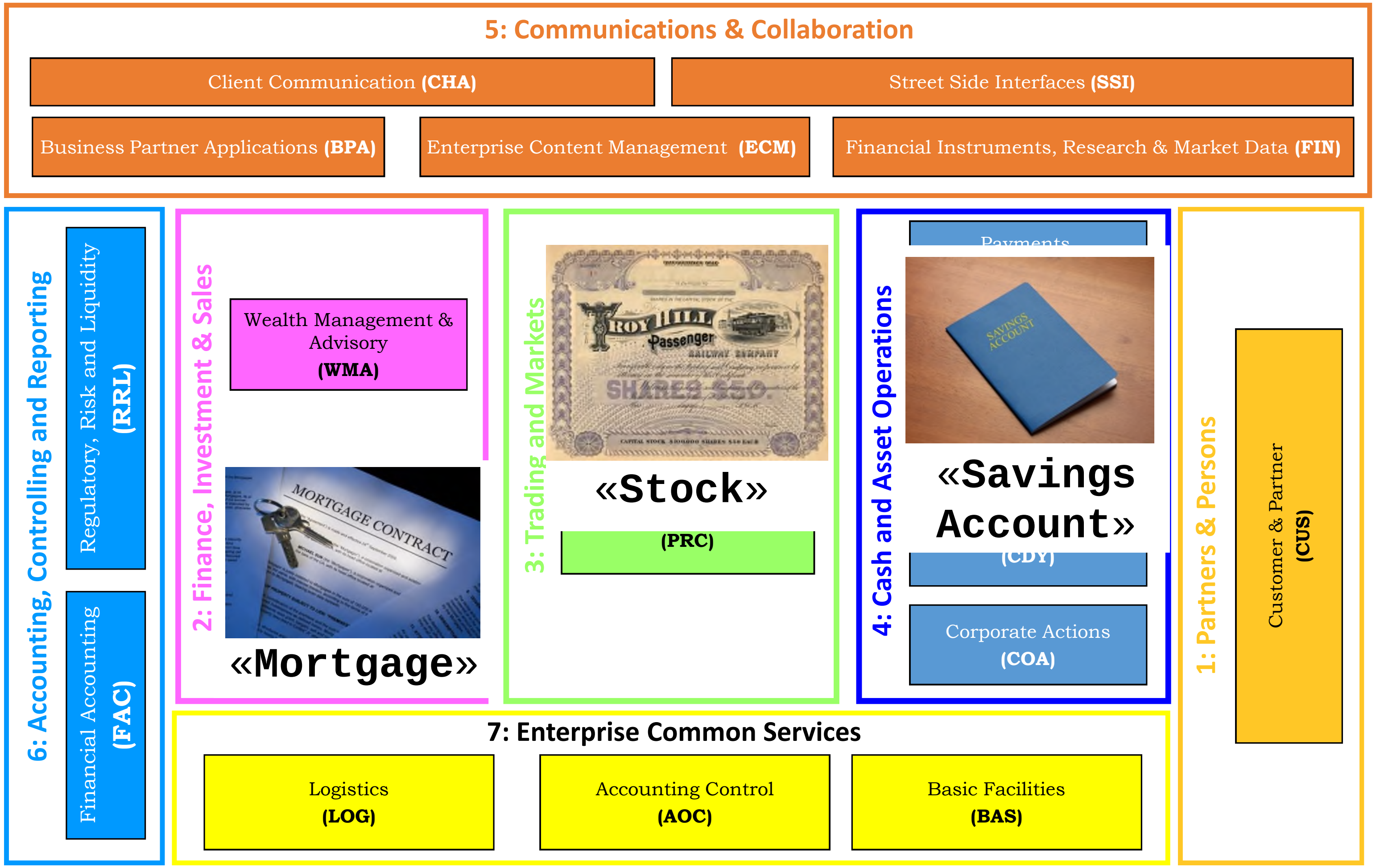


OWL (Web Ontology Language) Representation:

```
<owl:Class rdf:ID="Car"/>
<owl:Class rdf:ID="Body">
     <rdfs:subClassOf rdf:resource="Car"/>
</owl:Class>
<owl:Class rdf:ID="Chassis">
     <rdfs:subClassOf rdf:resource="Car"/>
</owl:Class>
<owl:Class rdf:ID="PowerTrain">
     <rdfs:subClassOf rdf:resource="Car"/>
</owl:Class>
```

How do we express **Semantics**?

Domain Model



The concepts must be clearly defined for all partners: This is done in the *domain model*

What is needed for full semantics?

Applications Interoperability

Collaborating applications must share a **common conceptual model (= Domain Model)**

Applications Interoperability

Semantic Interoperability

Definition of the concepts and their relationships → **Ontology**

Syntactic Interoperability

Technical Interoperability

What is needed for full semantics?

Collaborating applications must share a common conceptual model

Semantic Interoperability

Ontology

Syntactic Interoperability

Technical Interoperability

We have now understood:

- Technical interoperability
- Syntactical interoperability
- Semantic interoperability
- Applications interoperability

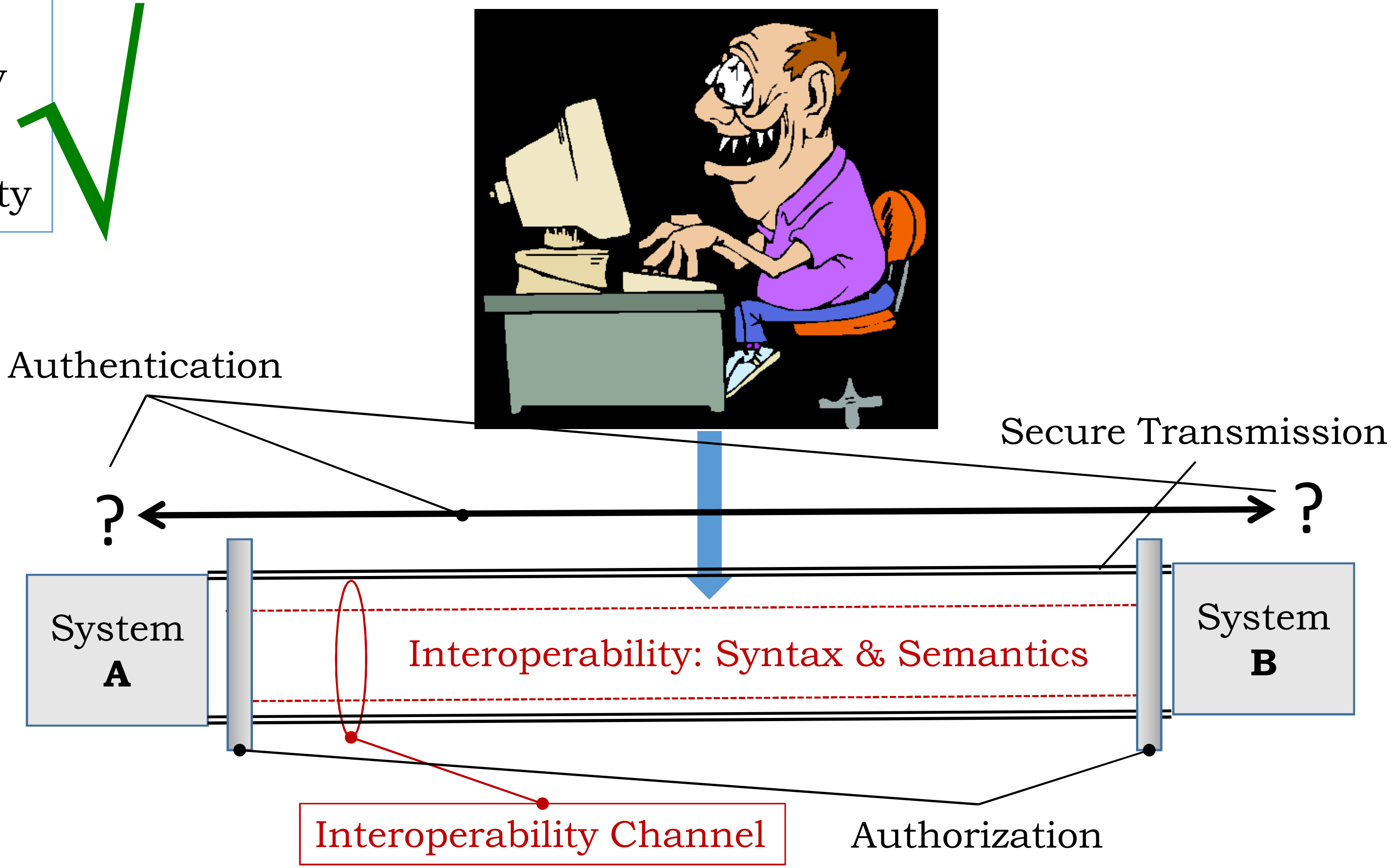
√

What happened to the **quality properties?**

- Security?
- Safety?
- Integrity?
- …

⇒ Additional concerns

Example: Secure Interoperability



Orthogonality

Security Functionality:

Confidentiality,
Authentication,
Authorization,
Integrity,
…

Security functionality and exchange functionality are **orthogonal**:

Never mix the two types of functionality!

- not in models
- not in architecture
- not in design
- not in implementation

Information & Control Exchange Functionality:
Technical interoperability
Syntactic interoperability
Semantic interoperability
Applications interoperability

A5

Architecture Principle A5:

Interoperability

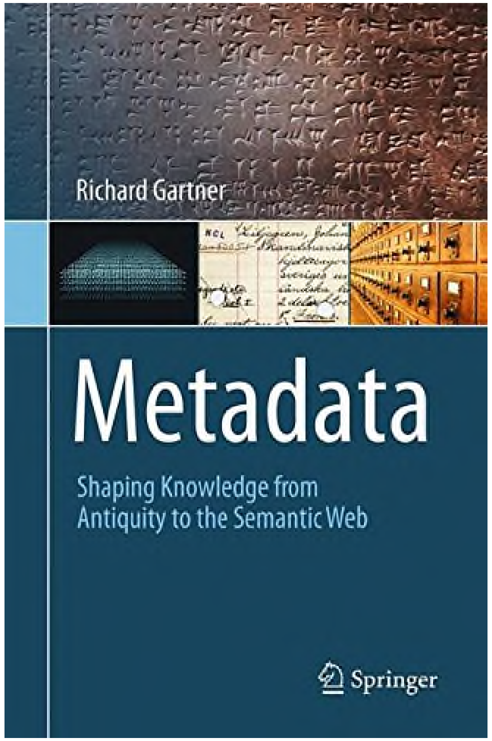
1. Precisely (formally) specify syntax and semantics in all interoperations
2. Whenever possible use formal contracts for the definition of interfaces
3. Whenever possible adopt and enforce accepted interoperability industry standards

Justification: Successful, unambigous interoperability is a key factor in today's distributed systems. Interoperabilty failures have severe consequences and are difficult to pinpoint. Formal contracts isolate the parts of the system.

Textbook

Tomas Erl et. al:
Web Service Contract Design and Versioning for SOA
Prentice Hall, Inc., USA, 2008. ISBN 978-0-136-13517-3

Textbook



Richard Gartner
Metadata – *Shaping Knowledge from Antiquity to the Semantic Web*
Springer-Verlag, Germany, 2016. ISBN 978-3-319-40891-0

A6

Architecture Principle A6:

Common Functions (X-functions and X-data)

Horizontal Architecture Layer Principles:

- A1: Architecture Layer Isolation
- A2: Partitioning, Encapsulation and Coupling
- A3: Conceptual Integrity
- A4: Redundancy
- A5: Interoperability
- A6: Common Functions
- A7: Reference Architectures, Frameworks and Patterns
- A8: Reuse and Parametrization
- A9: Industry Standards
- A10: Information Architecture
- A11: Formal Modeling
- A12: Complexity and Simplification

A disturbing **dilemma**:

A2
Partitioning

A4
Redundancy

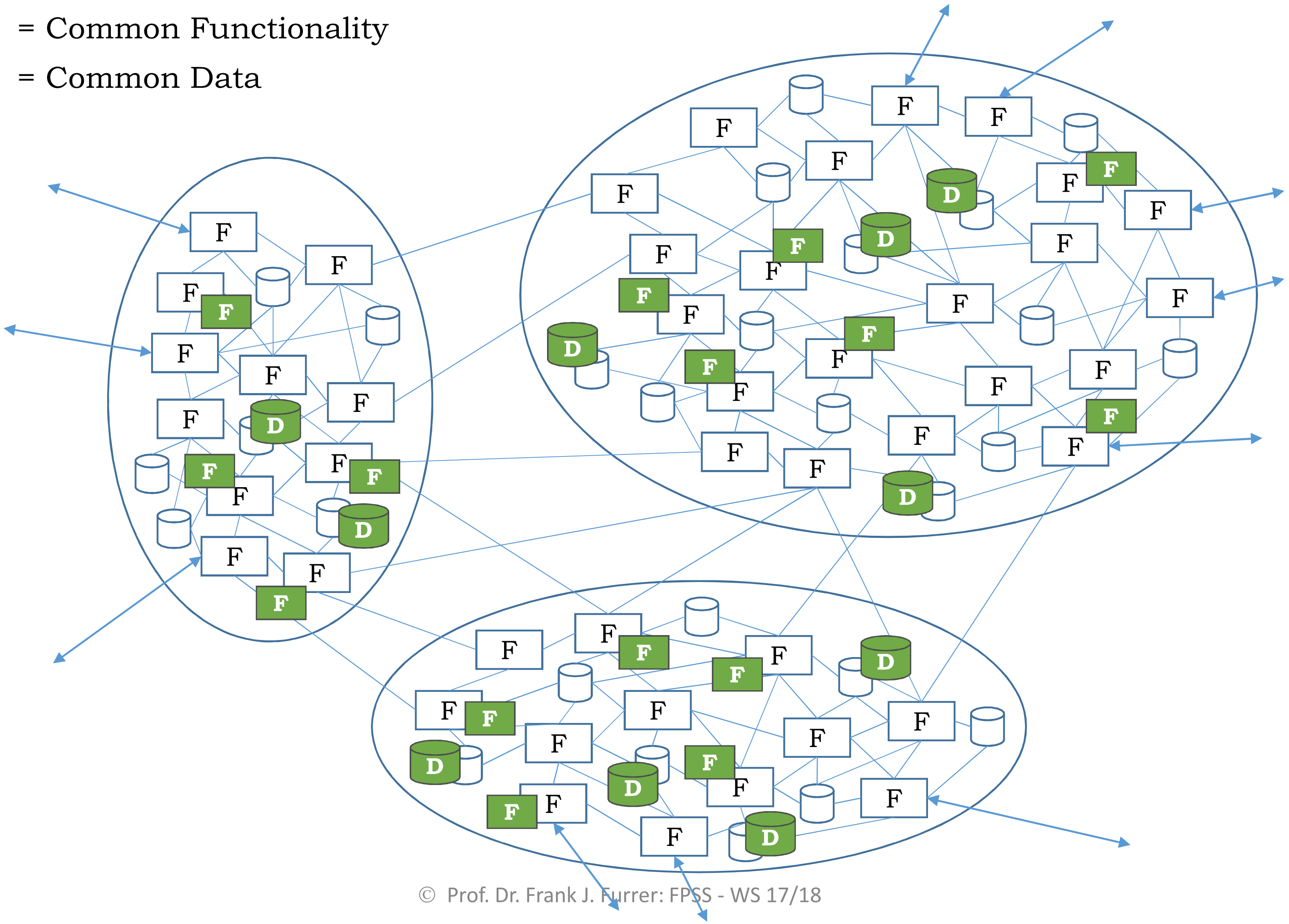
- Assign functionality & data to the **single** correct partition
- No unmanaged redundancy

What do we do if we need the *same* functionality or data in several partitions?

http://www.txcscopereview.com

F = Common Functionality

D = Common Data



F = Common Functionality   D = Common Data

DEFINITIONS

Common Functionality & Common Data/Information:

Functions or Data which are used in many parts of the system (and in <u>different</u> encapsulation units)

Danger:

- Break the partitioning rule (each function and data → exactly one partition)
- Generate unmanaged redundancy → divergence, inconsistency
- Risk performance problems → slow down, single points of failure

http://gallery.yopriceville.com

CAUTION: Common functions can infiltrate your system unnoticed!
… and they will

Example:

Programmer Action

Specification A

Specification B

Nearly identical specifications

Program Module A

http://creattica.com

Implementation **A**

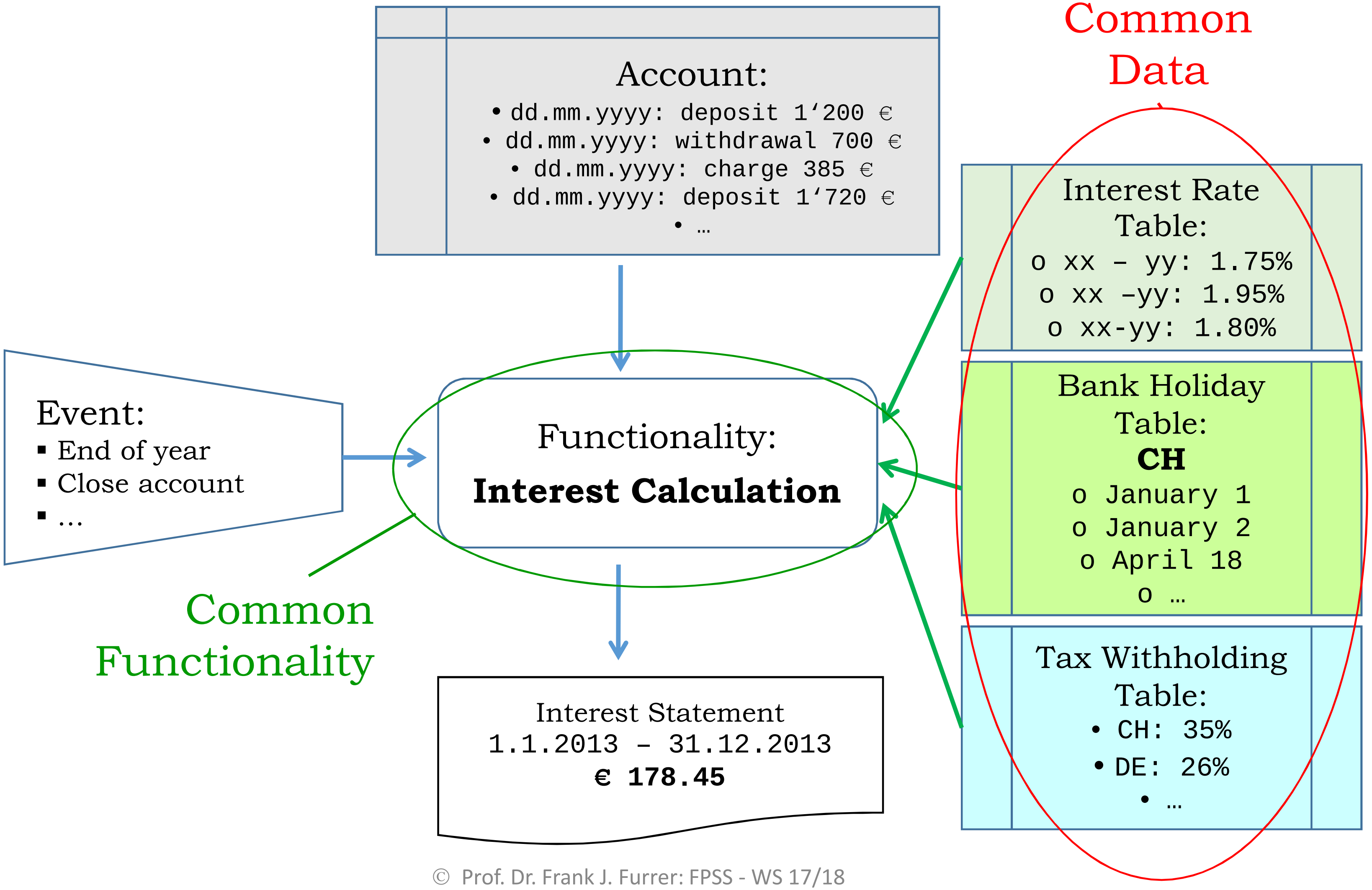
Program Module B

http://www.dreamstime.com

Implementation **B**

Example:

Interest Calculation

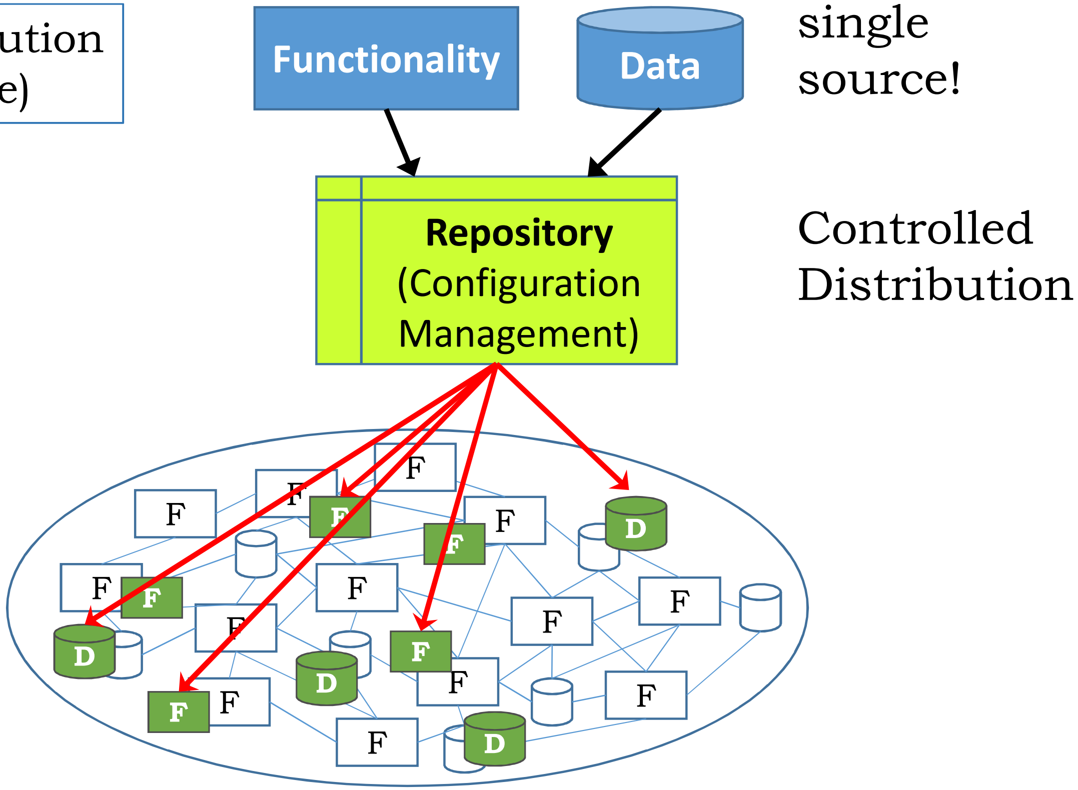


How can we deal with common functionality and data?

Identify, control and manage it!

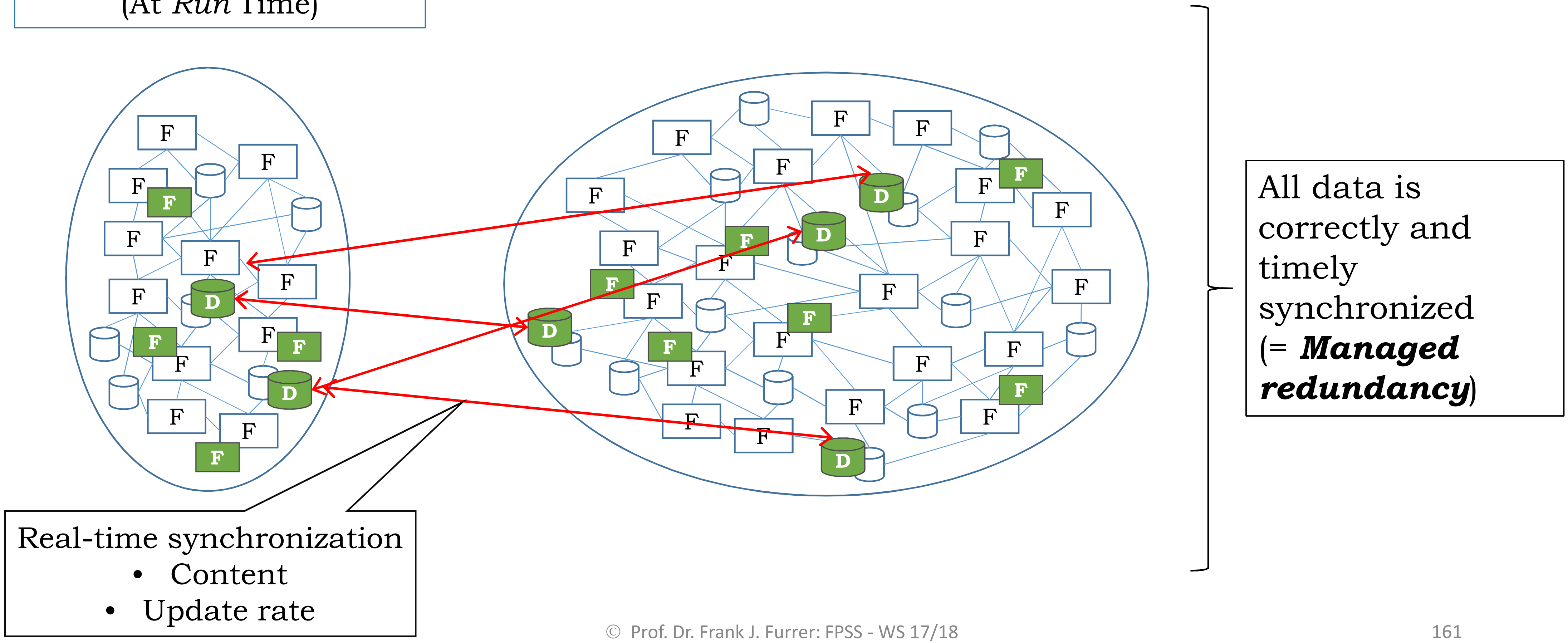
Managed Distribution (At *Build* Time)

It is exactly known at all times which *common functionality* is located where in the system



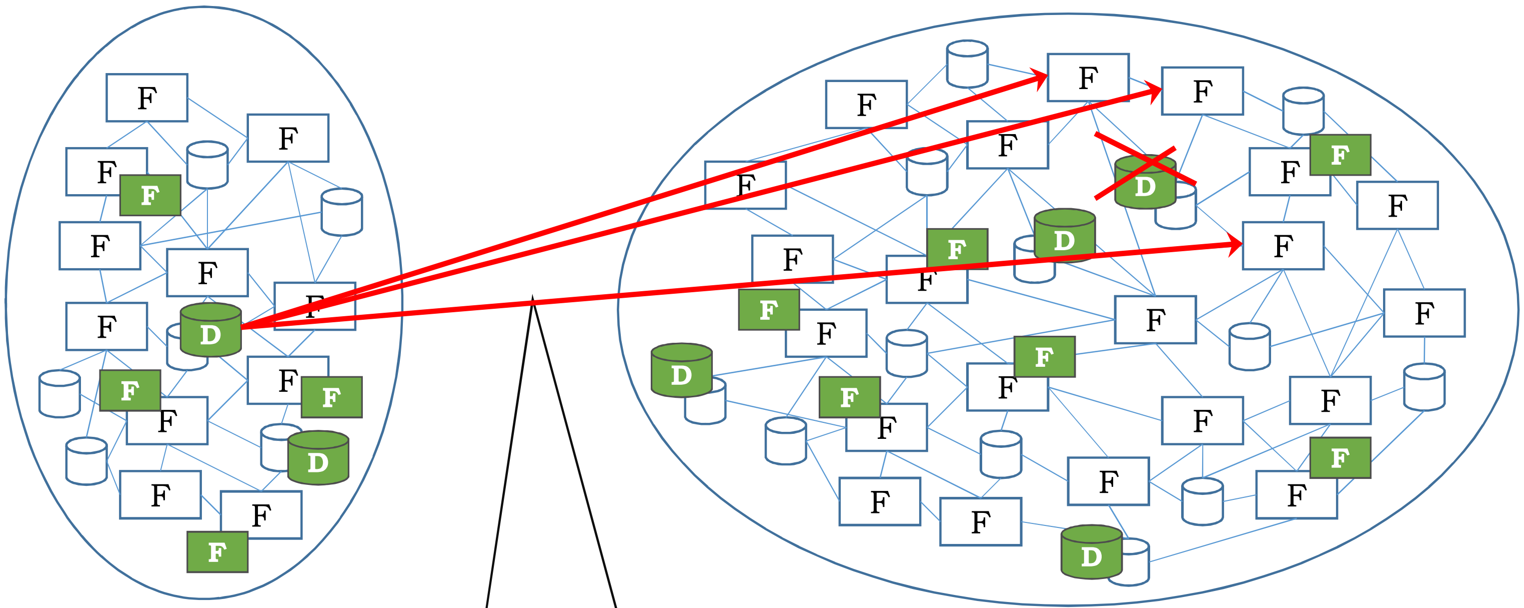
How can we deal with common functionality and data?

Managed Synchronization
(At *Run* Time)



How can we deal with common functionality and data?

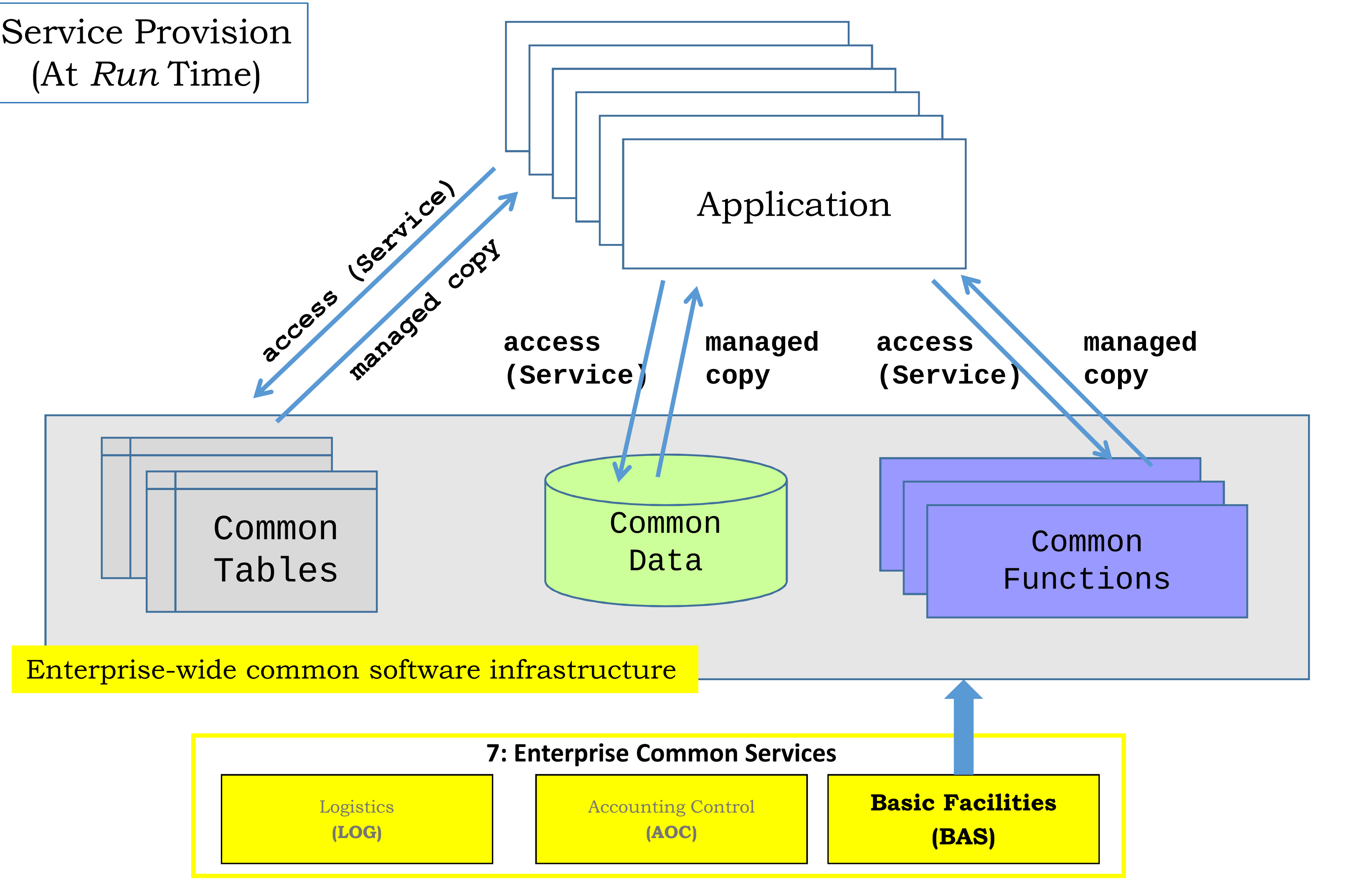
Service Provision
(At *Run* Time)



Provide Access Services

Common functionality and data are provided via ***services***

Service Provision
(At *Run* Time)



Common functionality and data are provided via a ***enterprise-wide software infrastructure***

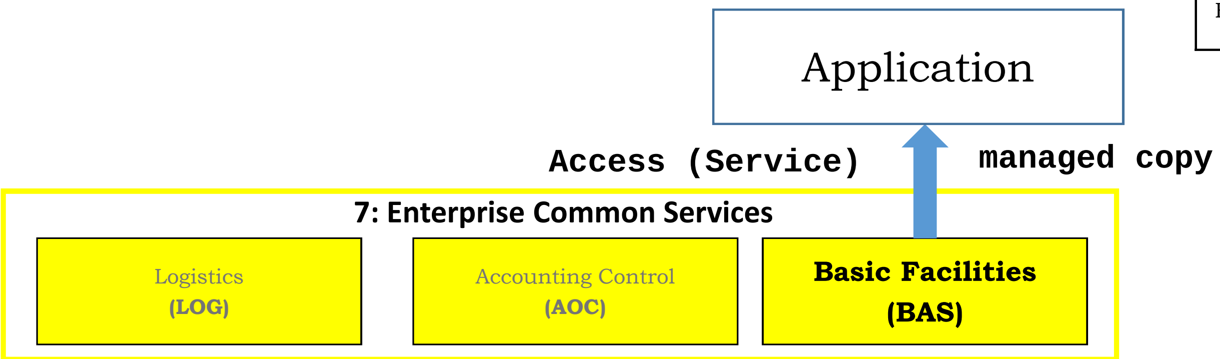
Example: Worldwide Bank Holidays

Common Data: Provide a list of bank holidays for <u>any country</u> and any year, such as 2017 for Cayman Islands

Provision: This function requires the maintenance of a (static) *table* containing all the bank holidays received from the respective local authorities

Bank Holiday 2012	Date
New Year's Day	Monday, January 2
National Heroes Day	Monday, January 23
Ash Wednesday	Wednesday, February 22
Good Friday	Friday, April 6
Easter Monday	Monday, April 9
Discovery Day	Monday, May 21
Queen's Diamond Jubilee	Monday, June 4
Queen's Anniversary	Monday, June 18
Constitution Day	Monday, July 2
Public Holiday	Wednesday, July 18
Remembrance Day	Monday, November 12
Christmas Day	Tuesday, December 25
Boxing Day/Family Day	Wednesday, December 31

List of Cayman bank holidays 2012
http://www.bank-holidays.com



A6

Architecture Principle A6:

Common Functions

1. Identify all common functions and common data (= cross-cutting concerns in an IT-architecture)
2. Provide managed solutions to all cross-cutting concerns, avoiding unmanaged redundacy
3. Whenever possible provide and enforce a company-wide software-infrastructure

Justification: Cross-cutting concerns (Common functions and data) have a high inherent risk to diverge and thus cause unmanged reduncancy or inconsistent implementations – which can be an unknown and serious danger to an IT-system (especially a large or very large IT-system)

Remember

Our objective is:

To build, evolve, and maintain long-lived, mission-critical IT-systems with a strong dependability, an easy changeability, and a high business value.

A1 ...

A12

http://www.gettyimages.ch

Part 3A: A1 – A6