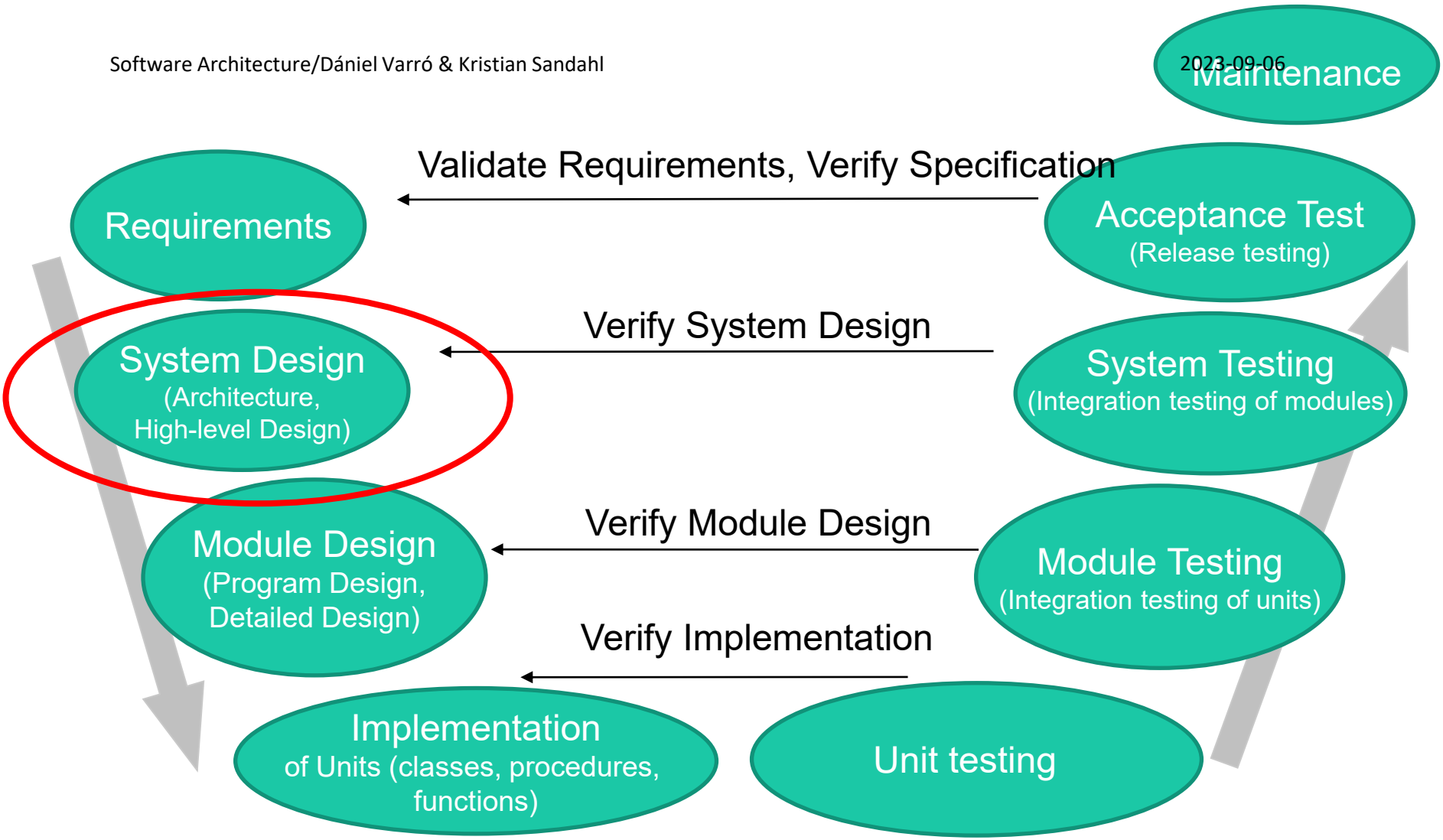
Software Architecture

Dániel Varró / Kristian Sandahl

LINKÖPING UNIVERSITY


Maintenance
Validate Requirements, Verify Specification
Requirements
Acceptance Test
(Release testing)
Verify System Design
System Design
(Architecture,
High-level Design)
System Testing
(Integration testing of modules)
Verify Module Design
Module Design
(Program Design,
Detailed Design)
Module Testing
(Integration testing of units)
Verify Implementation
Implementation
of Units (classes, procedures,
functions)
Unit testing
Project Management, Software Quality Assurance (SQA), Supporting Tools, Education

Motivation for Architecture

Why should we design a system?

Carol
the customer

Requirements

Why not go directly?

Harry
the hacker

Implementation

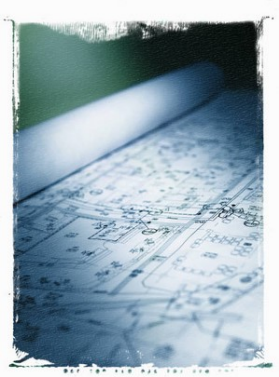
Constructing a building...

The king's requirements

Construction

Constructing a building...

The king's requirements



Architecture

Ulla

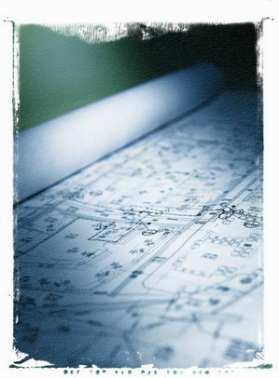
Construction

Constructing software...

Software is different

- No physical natural order of construction (e.g. start with the foundation of the house)
- Software is not tangible
- Sometimes a large semantic gap
- You need a map to coordinate efforts

That's not to say that customers and implementers should not meet!



Architecture

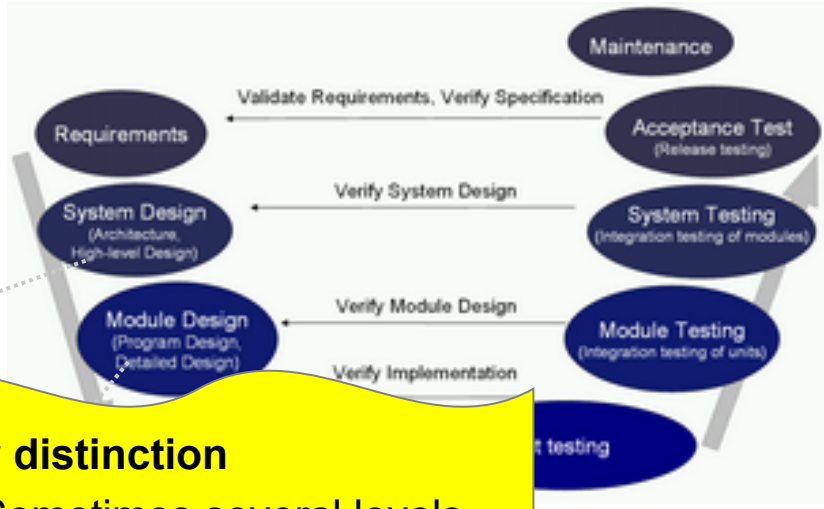
Carol
the customer

Requirements

Harry
the hacker

Implementation

Constructing software...



Abstraction

System Design
(Architecture,
High-level Design)

Module Design
(Program Design,
Detailed Design)

Fuzzy distinction

- Sometimes several levels
- Sometimes only one level

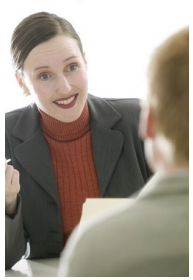
Carol
the customer

Harry
the hacker

Requirements

Implementation

Why design and document software architectures?



Communication between stakeholders

A high-level presentation of the system.

Use for understanding, negotiation and communication.

Early design decisions

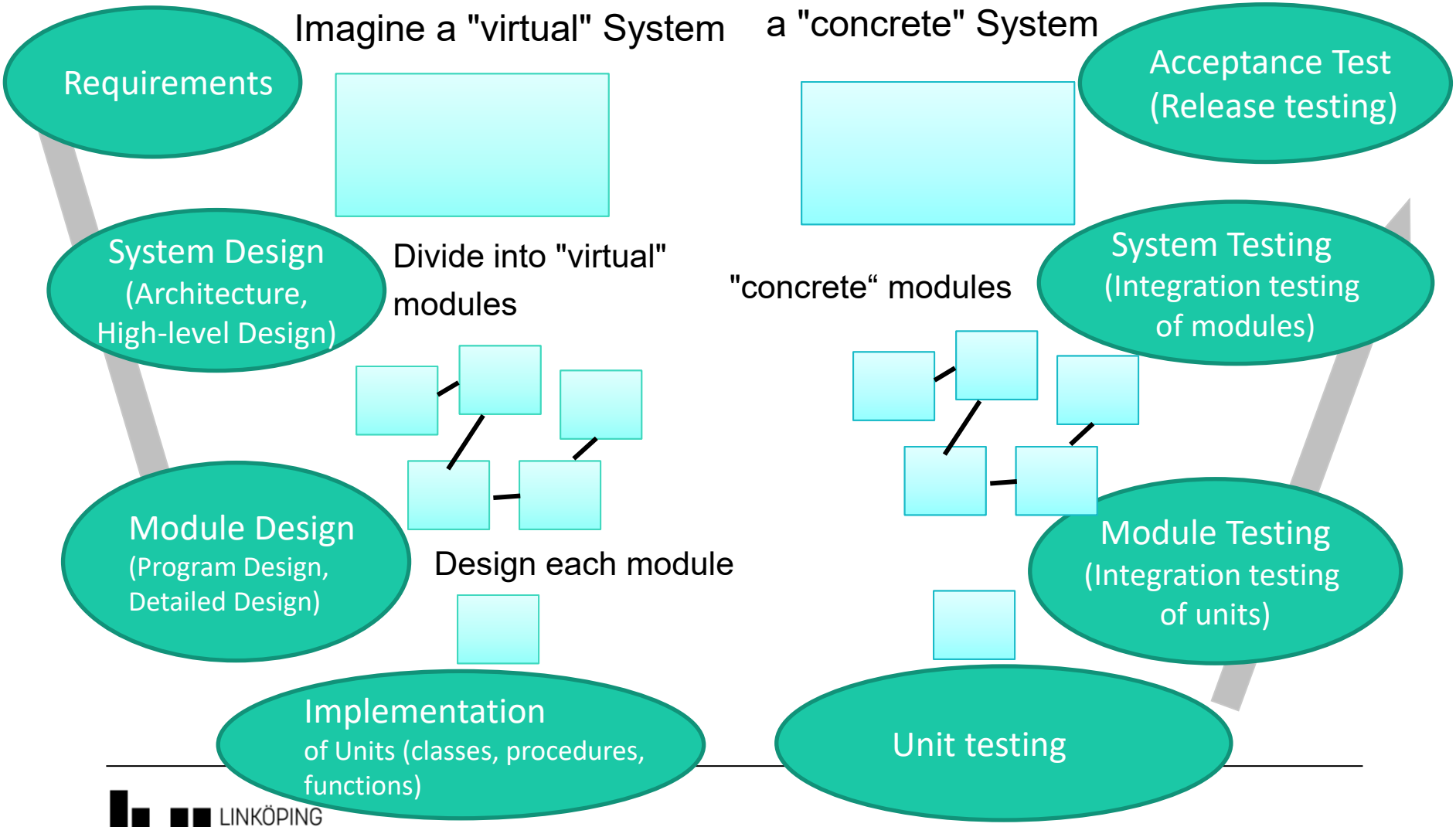
Profound effect on the systems quality attributes, e.g. performance, availability, maintainability etc.

Large-scale reuse

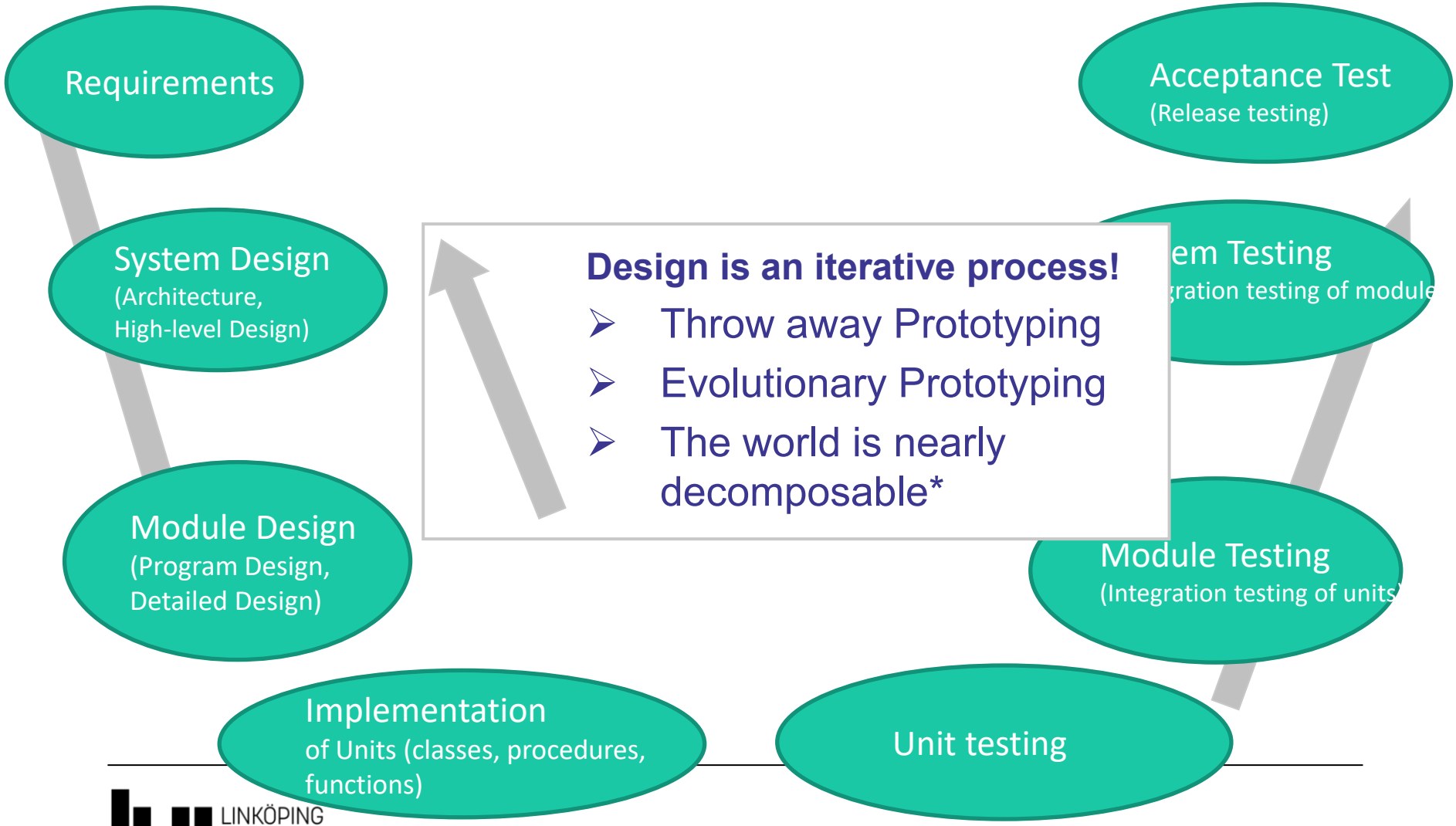
If similar system have common requirements, modules can be identified and reused. (Bass et.al., 2003)

System vs. Software Architecture: General Concepts and Views

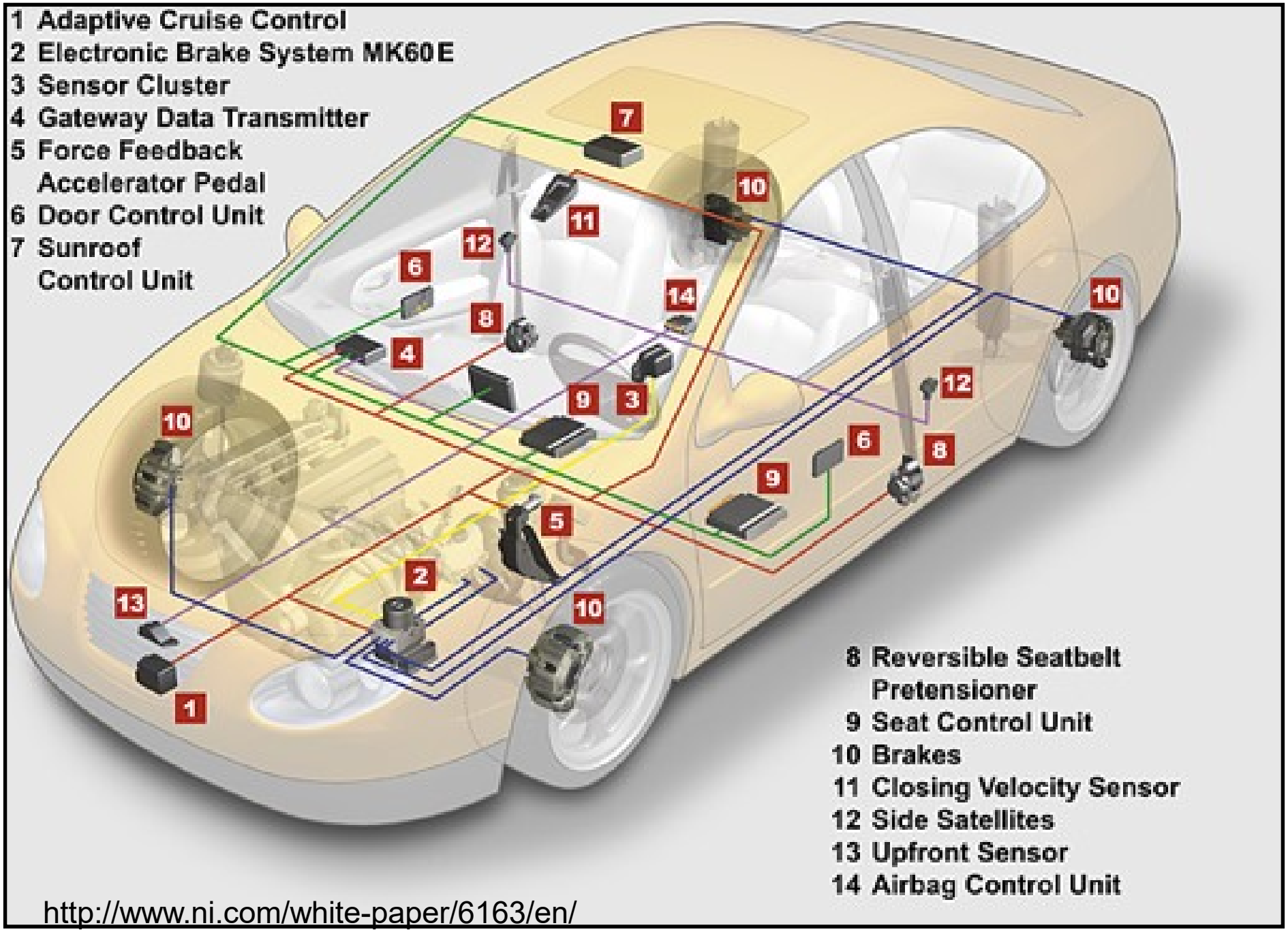
Analyze and Synthesis a system (decompose and compose)



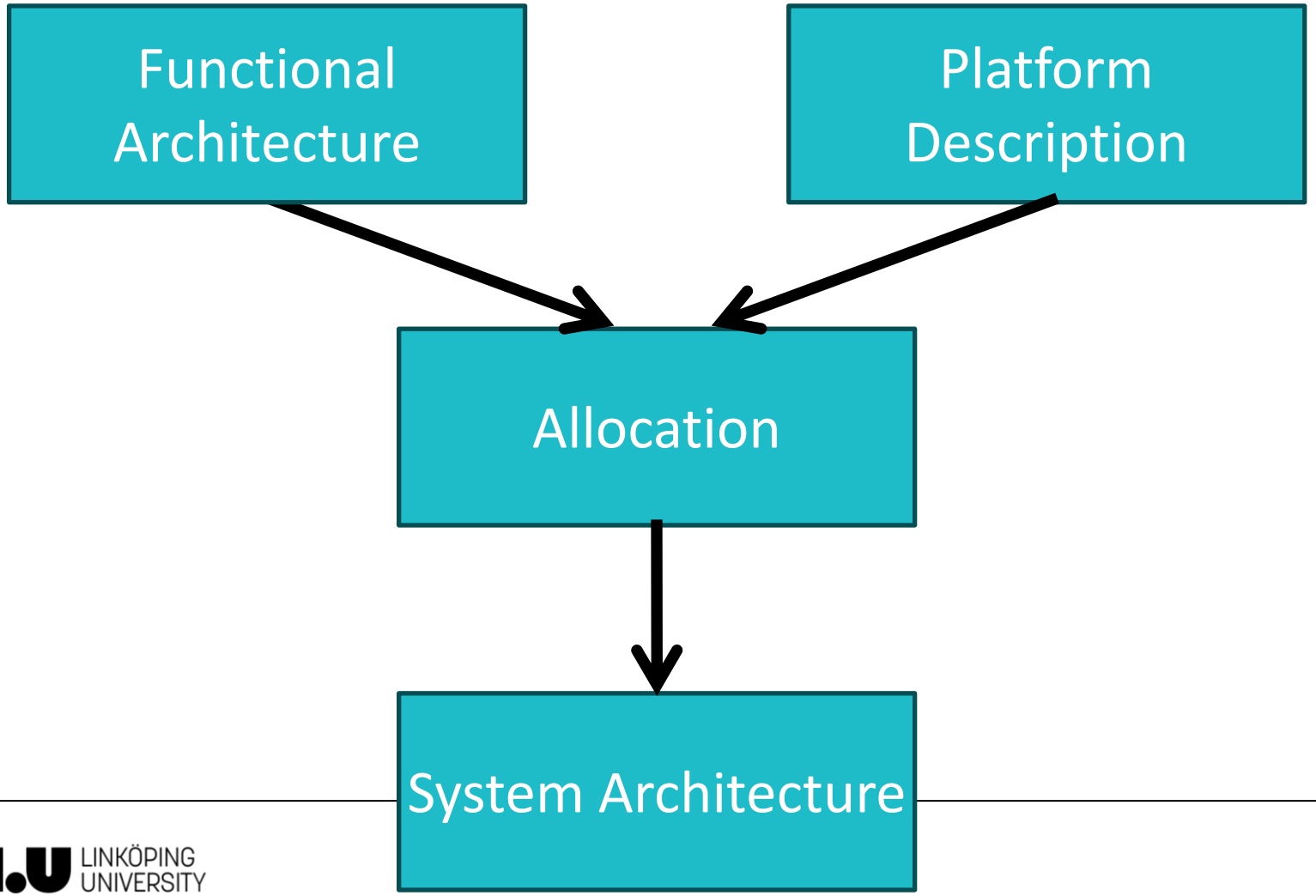
Analyze and Synthesis a system (decompose and compose)



*Herbert Simon


1 Adaptive Cruise Control
2 Electronic Brake System MK60E
3 Sensor Cluster
4 Gateway Data Transmitter
5 Force Feedback
Accelerator Pedal
6 Door Control Unit
7 Sunroof
Control Unit
8 Reversible Seatbelt
Pretensioner
9 Seat Control Unit
10 Brakes
11 Closing Velocity Sensor
12 Side Satellites
13 Upfront Sensor
14 Airbag Control Unit
http://www.ni.com/white-paper/6163/en/

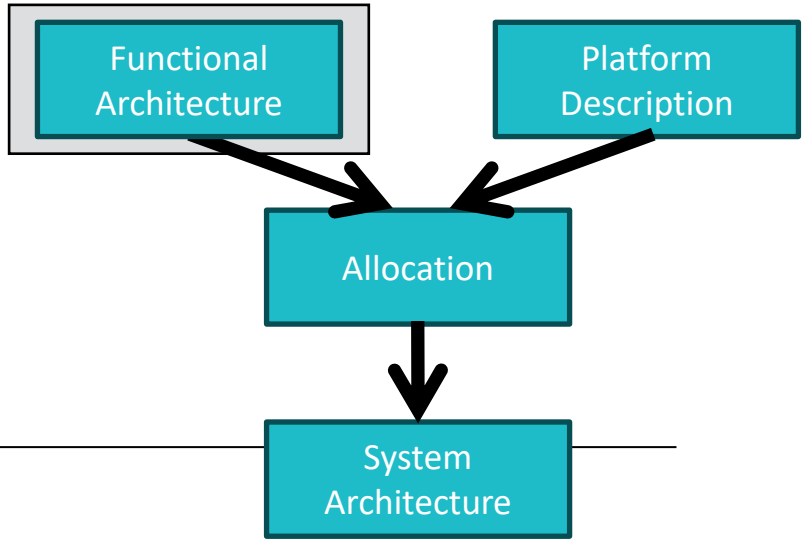
Overview of System Architecture



Functional / Logical Architecture

Functional (logical) decomposition of system into subsystems / components

- Component: Deployable & executable unit with precise interfaces at well-defined points of service
- Interfaces: Functionality, interaction



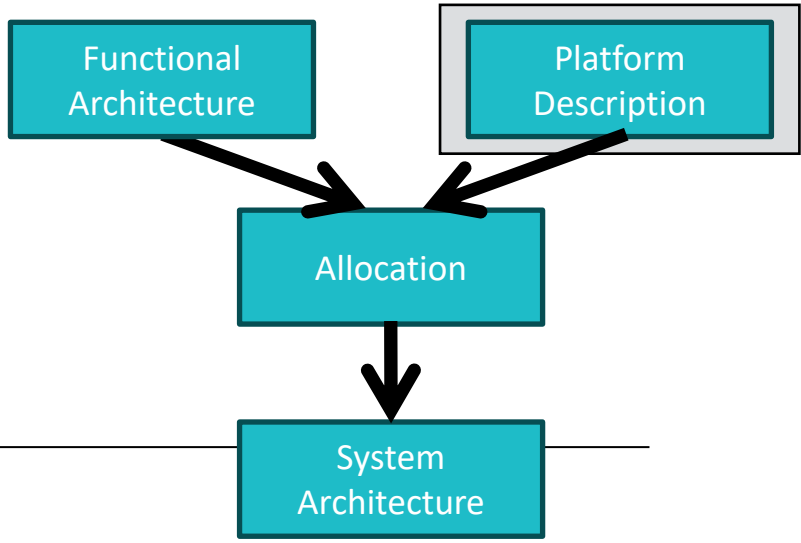
Platform Description

Specification of HW/SW platform:

- Nodes: Execution units (processors, ECUs)
- Their physical interconnection (e.g., buses, wires)

Examples:

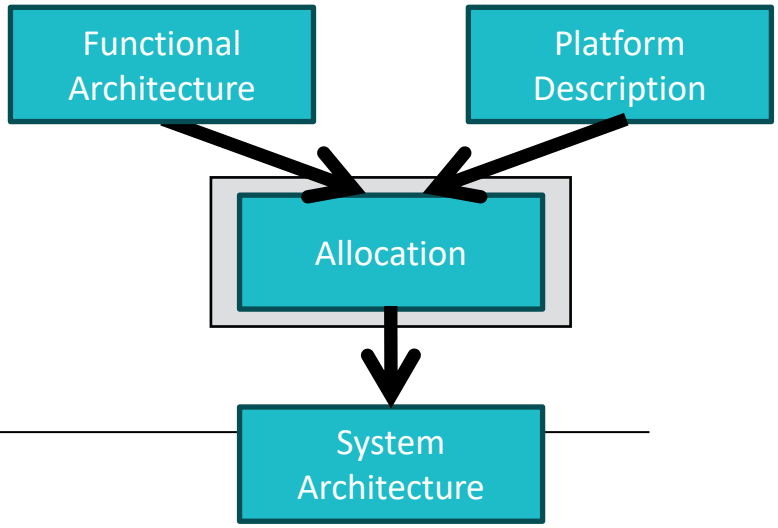
- AUTOSAR (automotive)
- ARINC 653 (avionics)
- Cloud providers, IT infra.



Allocation

Mapping of functional components to hardware/software platform by respecting:

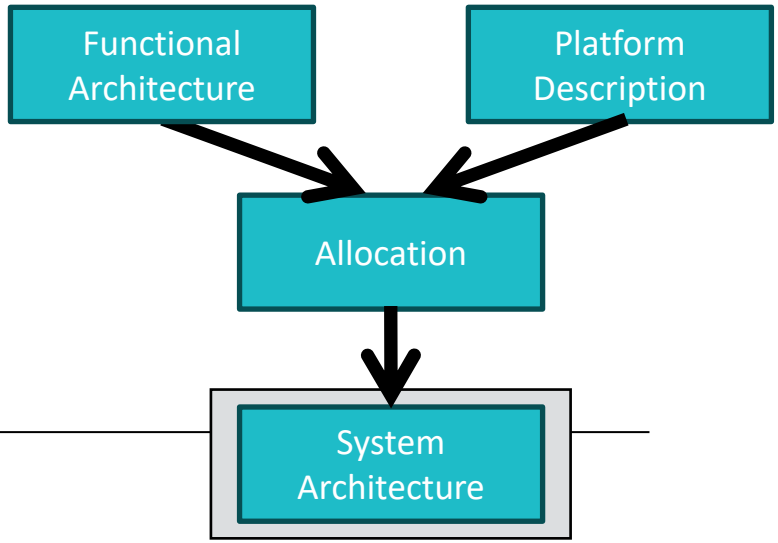
- Schedule, timeliness constraints
- Redundancy, fault-tolerance requirements
- Reliability, availability agreements
- Performance constraints



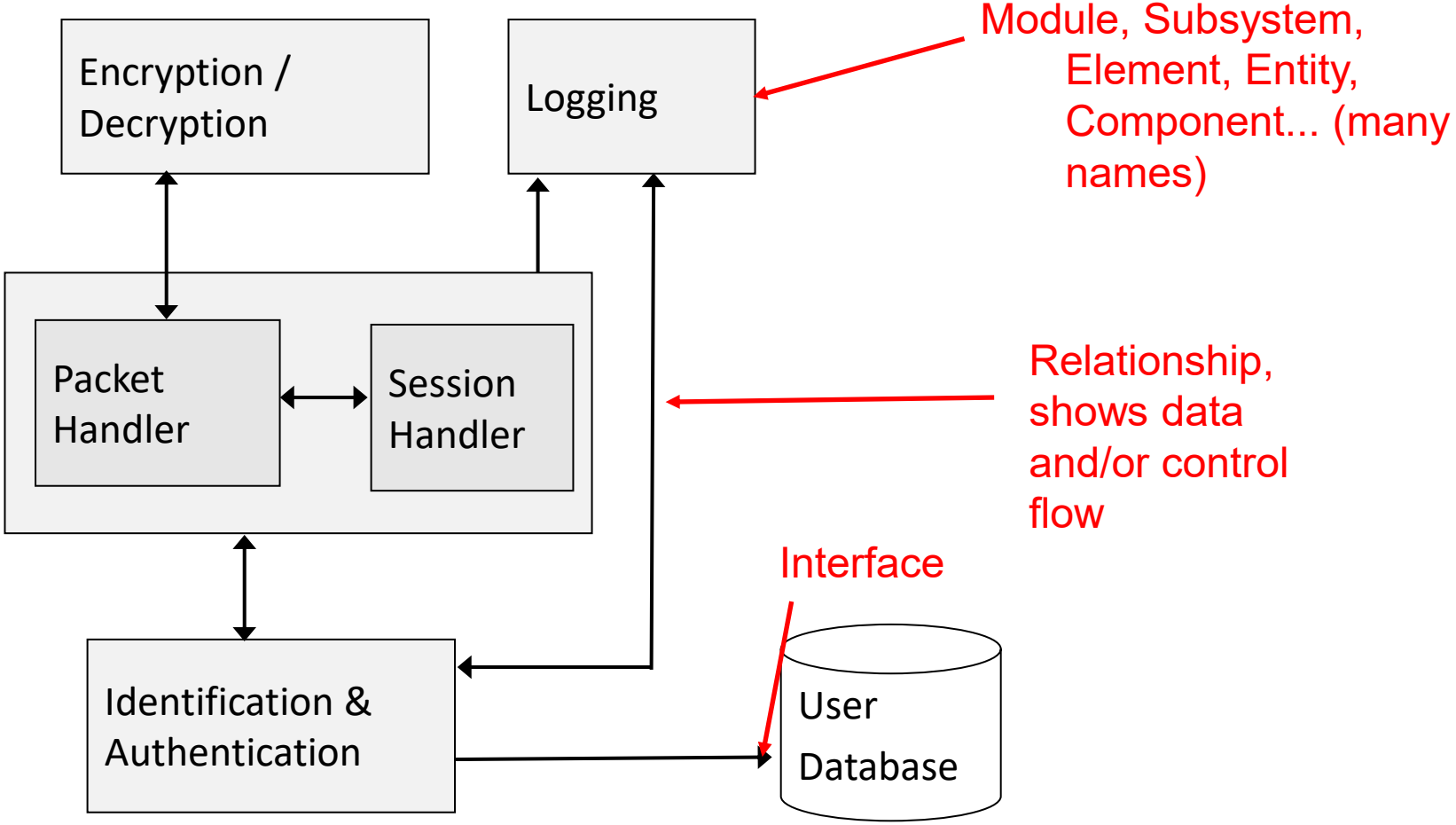
System Architecture

Result of the allocation step that:

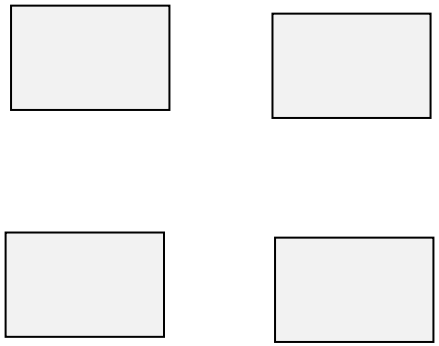
- Is ready for deployment
- Specifies or derives configuration files



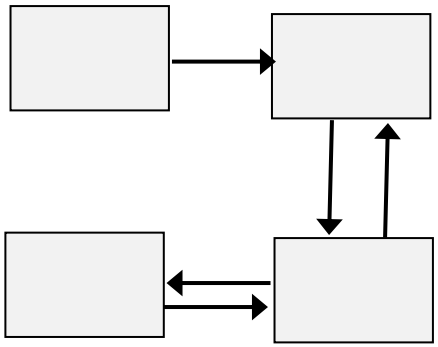
Block (Box-and-line) diagrams...



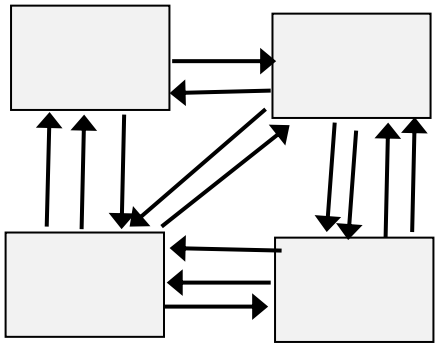
Coupling - dependency between modules



Uncoupled - no dependencies



Loosely coupled - few dependencies



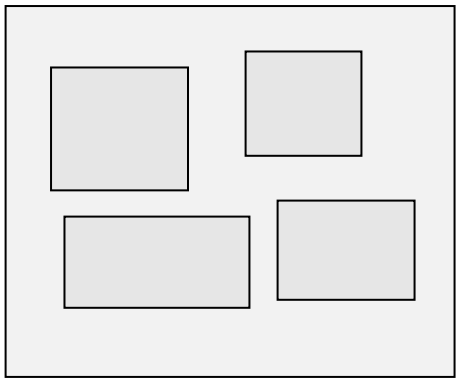
Highly coupled - many dependencies

What do we want?

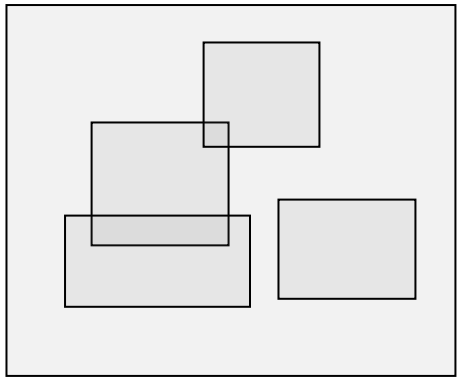
Low coupling. Why?

- Replaceable
- Enable changes
- Testable - isolate faults
- Understandable

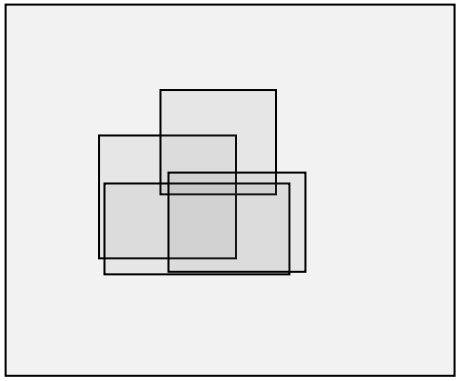
Cohesion - relation between internal parts of the module



Low cohesion - the parts e.g. functions have less or nothing in common.



Medium cohesion - some logically related function, e.g. I/O related functions



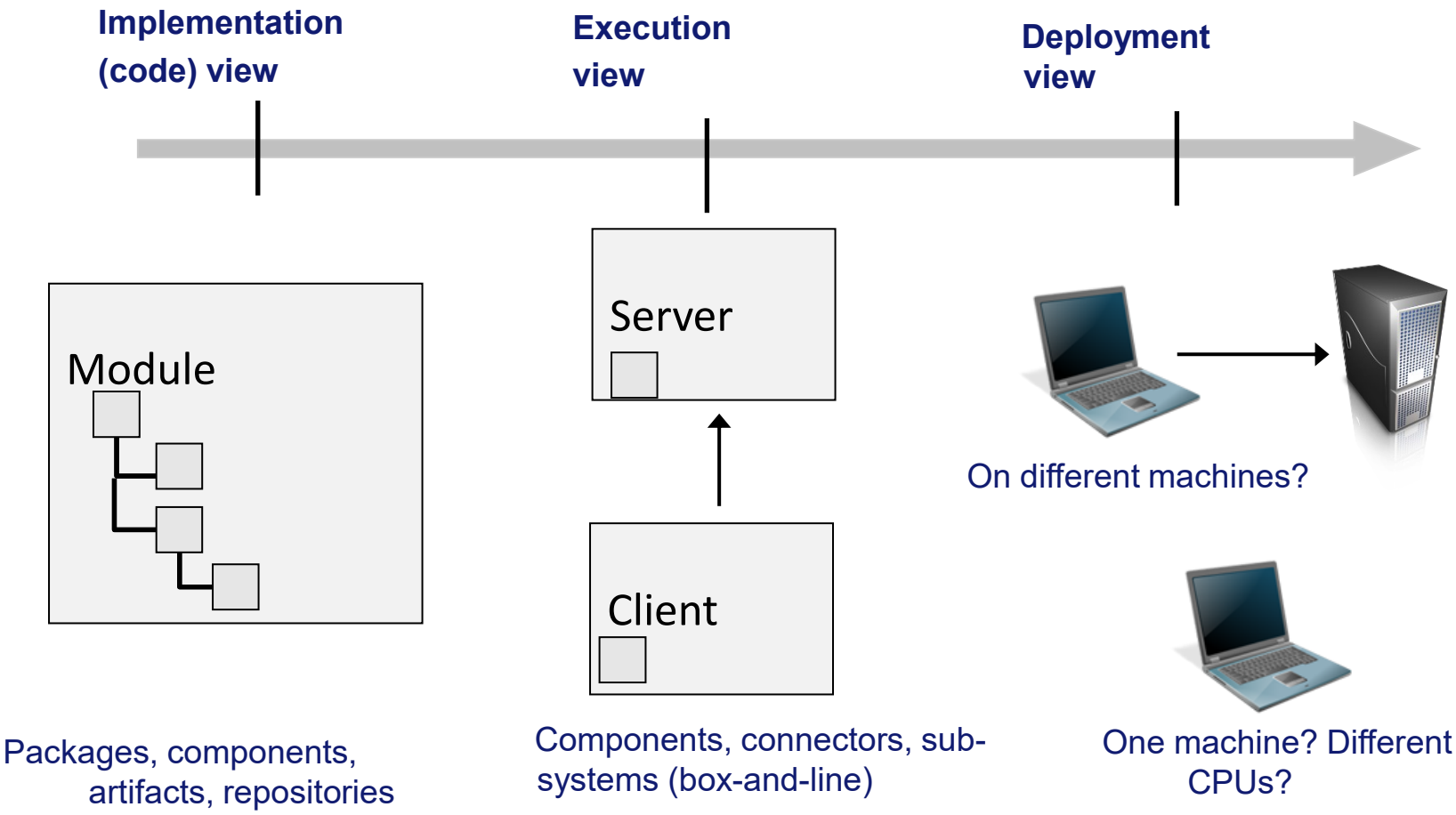
What do we want?

High cohesion. Why?

- More understandable
- Easier to maintain

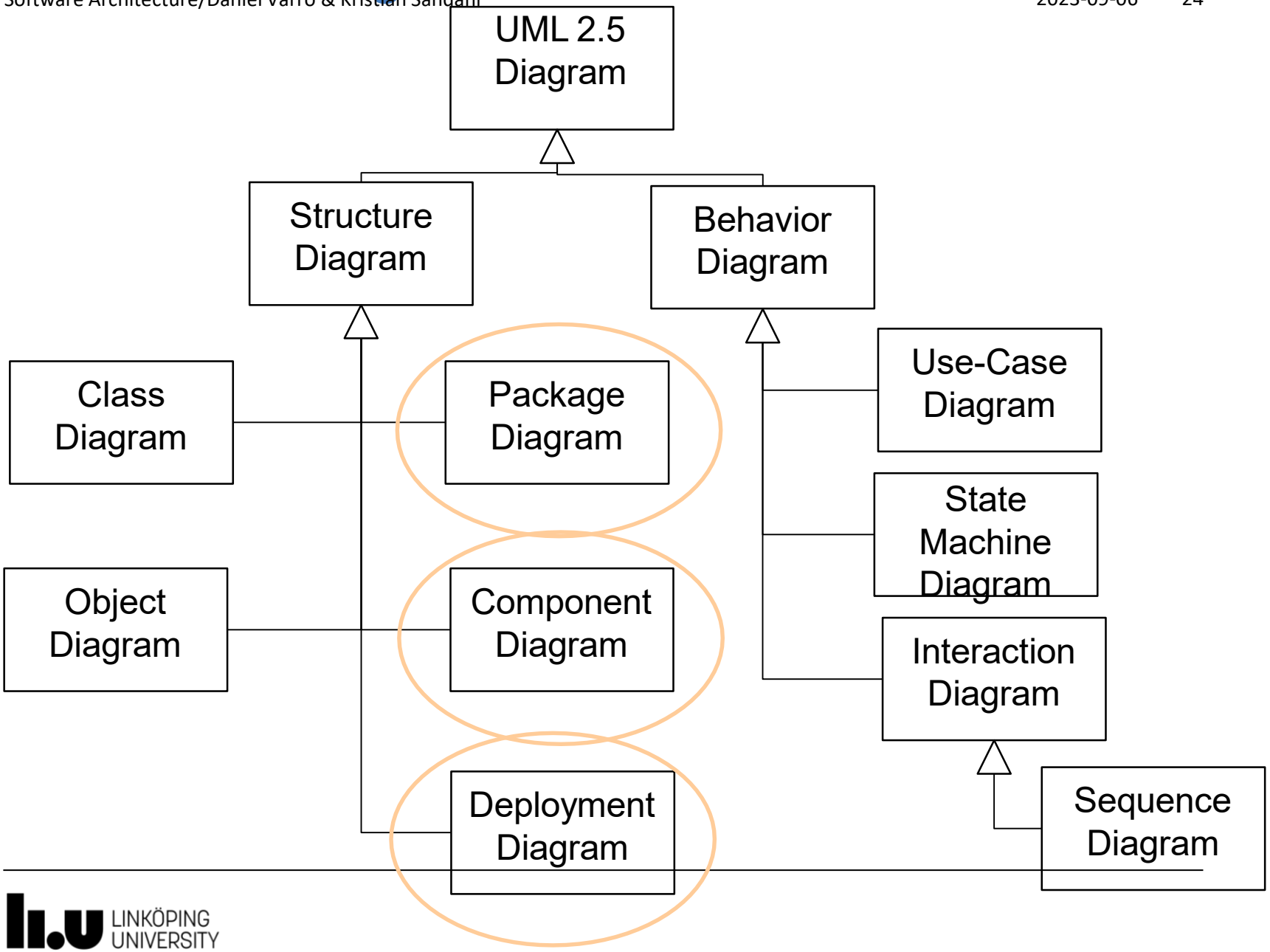
High cohesion - does only what it is designed for

Architectural views



Architecture Modeling in UML

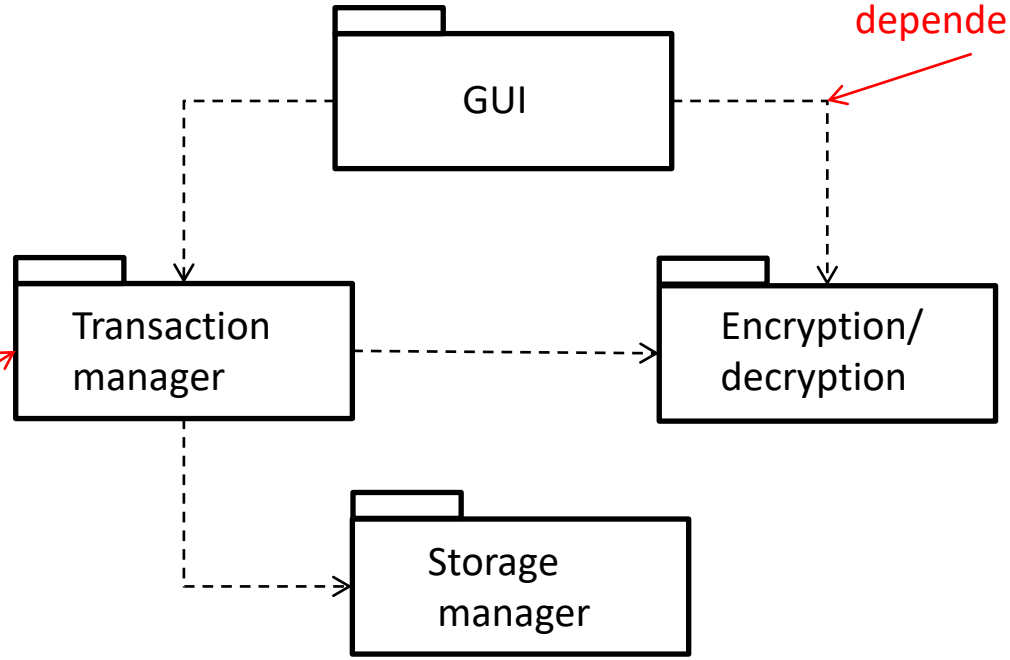
Well-known Diagrams of UML in architecture



Implementation view with packages

A developer's perspective:

1. What are we going to develop?
2. Where is the code?

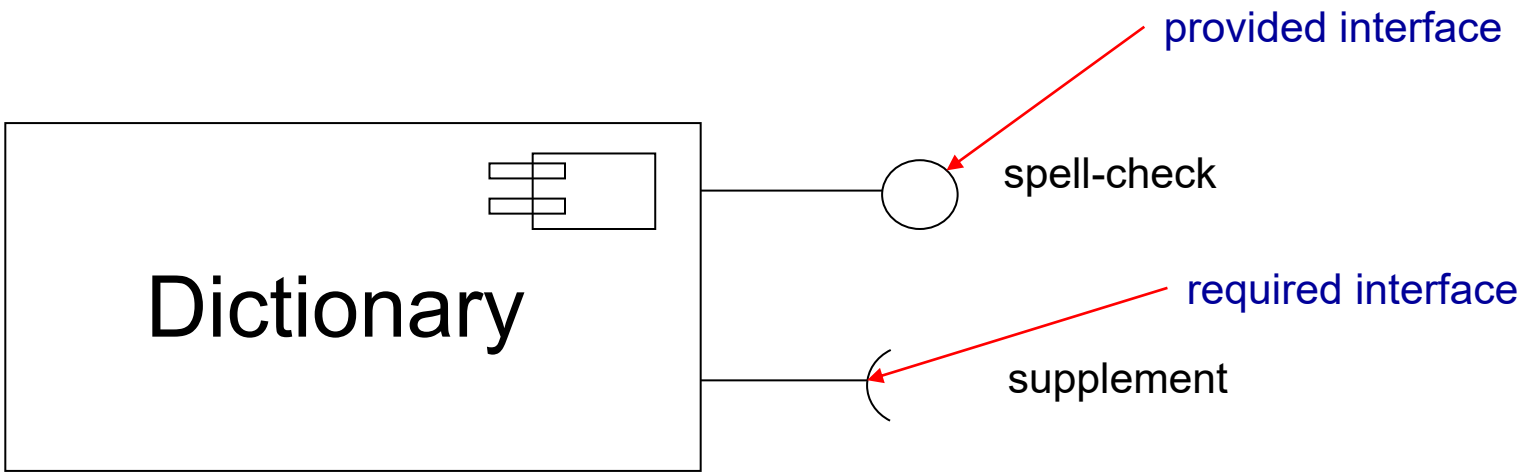


Package

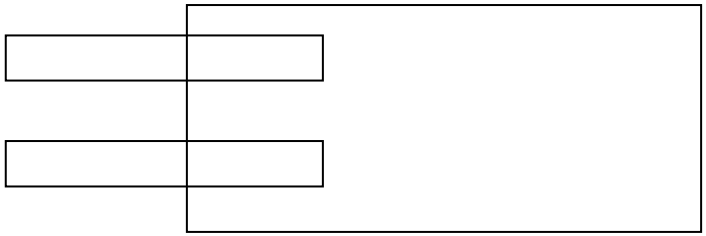
- Organize work
- Compile together
- Name space

Packages can be used to give an overall structure to other things than code, eg. Use-cases and Classes

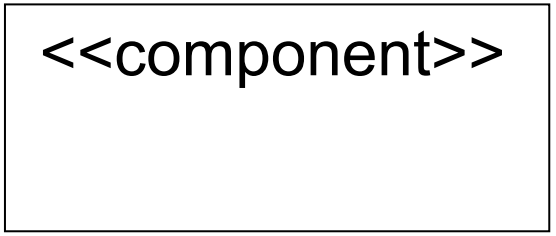
Component diagram with interfaces



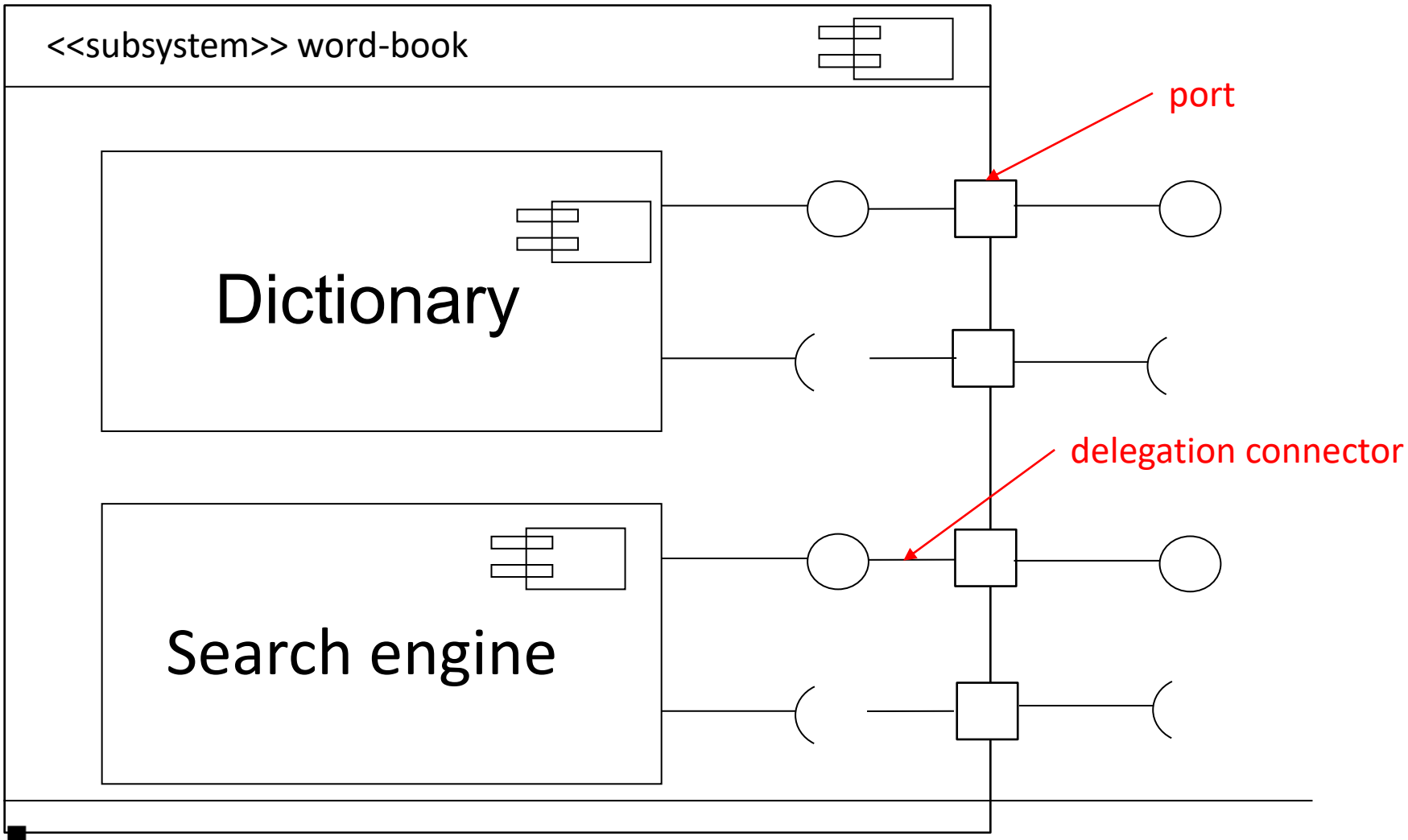
Older notation:



Alternative notation:

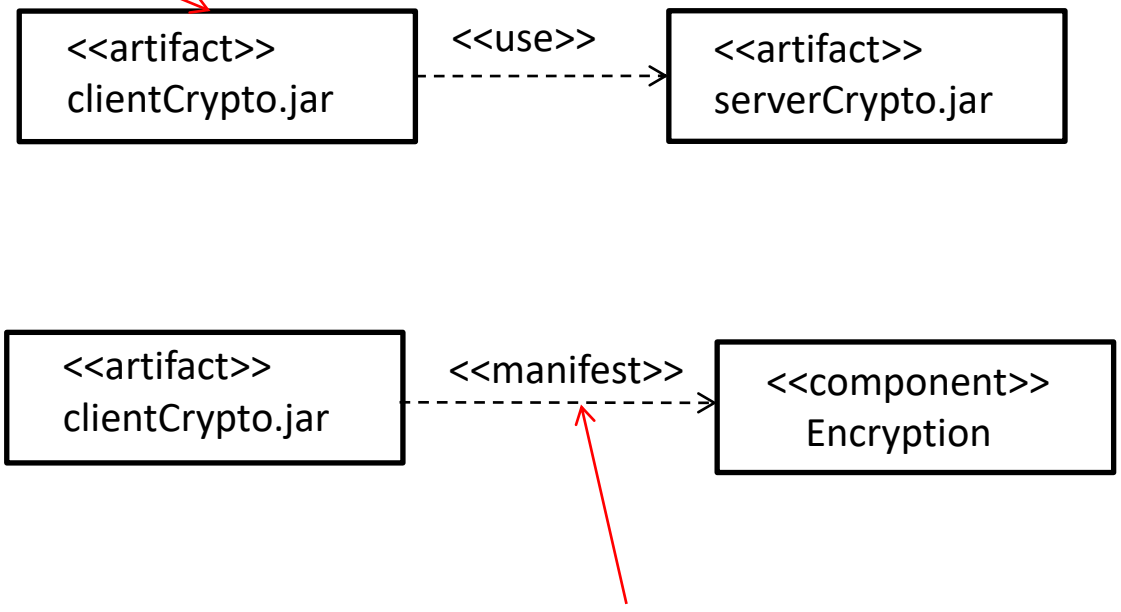


Subsystem with components

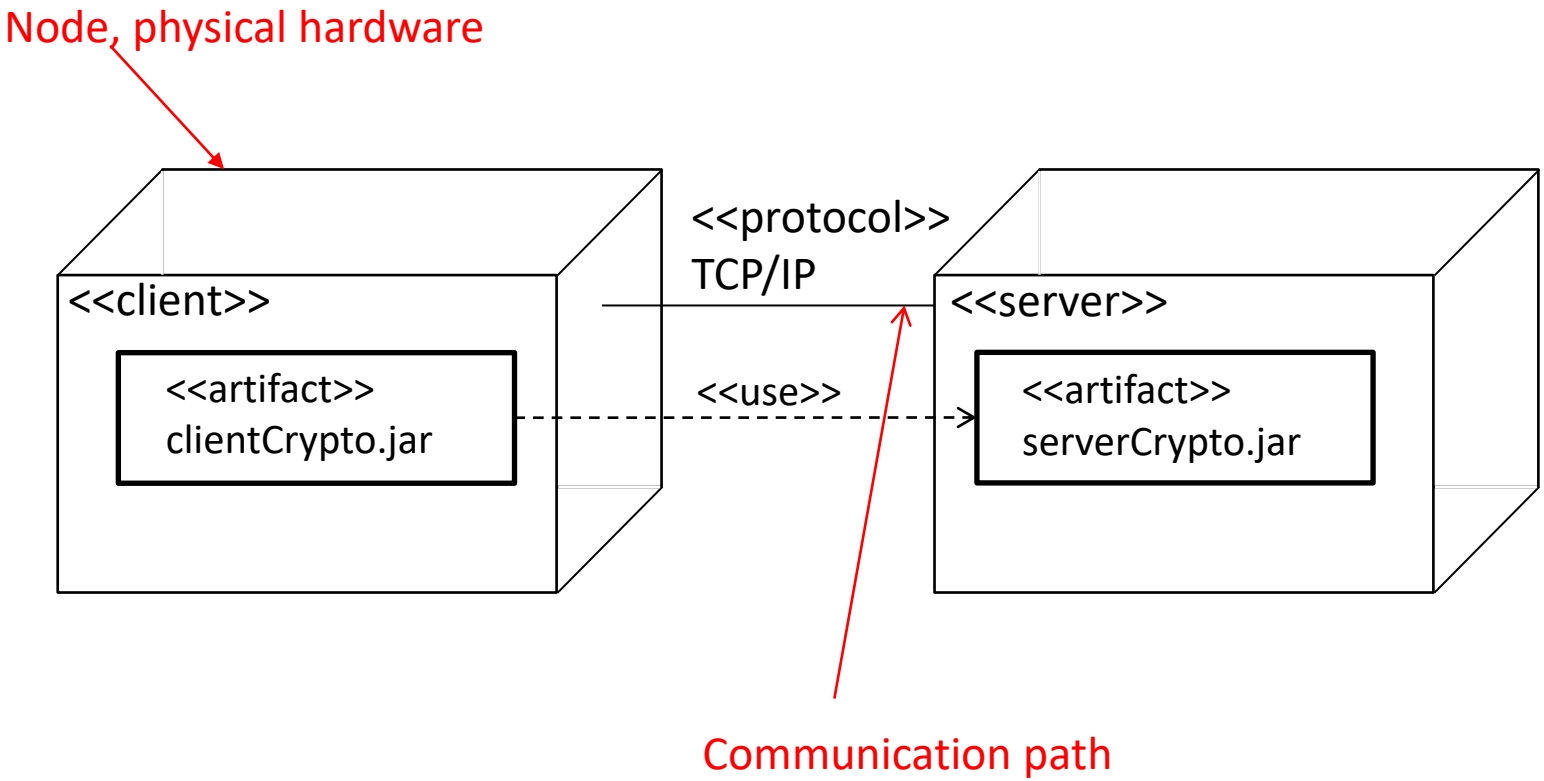


Artifacts

Physical code, file, or library



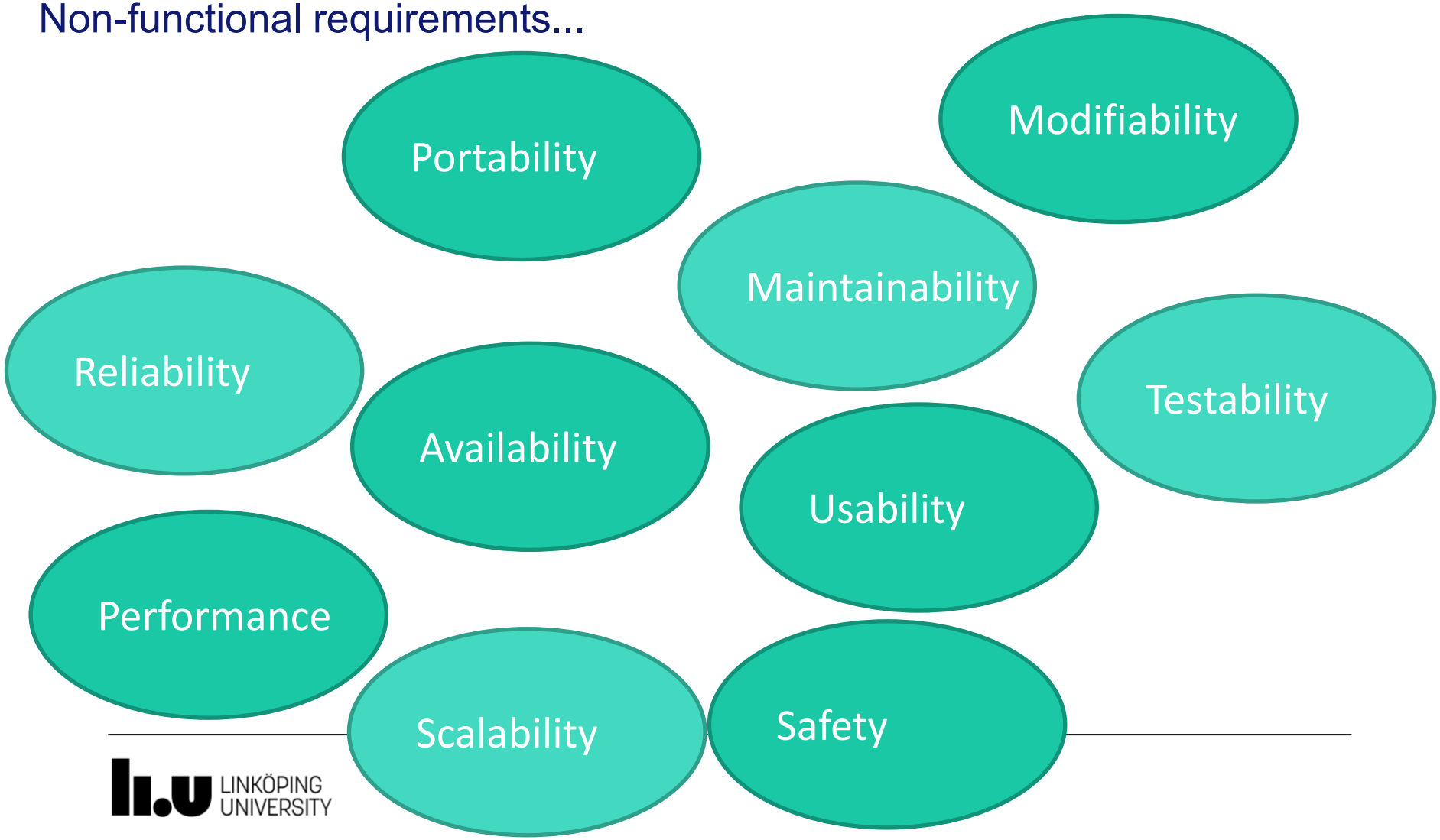
Deployment view in UML



Architecture and Quality Factors

Several quality factors - sometimes overlap

Non-functional requirements...



How can you design a system for better performance?

- Throughput
- Response time in an interactive system

Performance

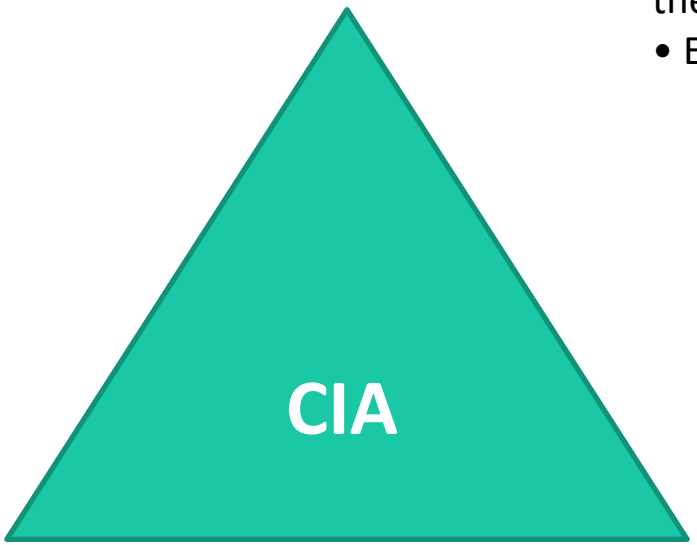
Scale up...

Scale out...

How can you design a secure system

Confidentiality

- Only authorized users can read the information
- E.g. Military



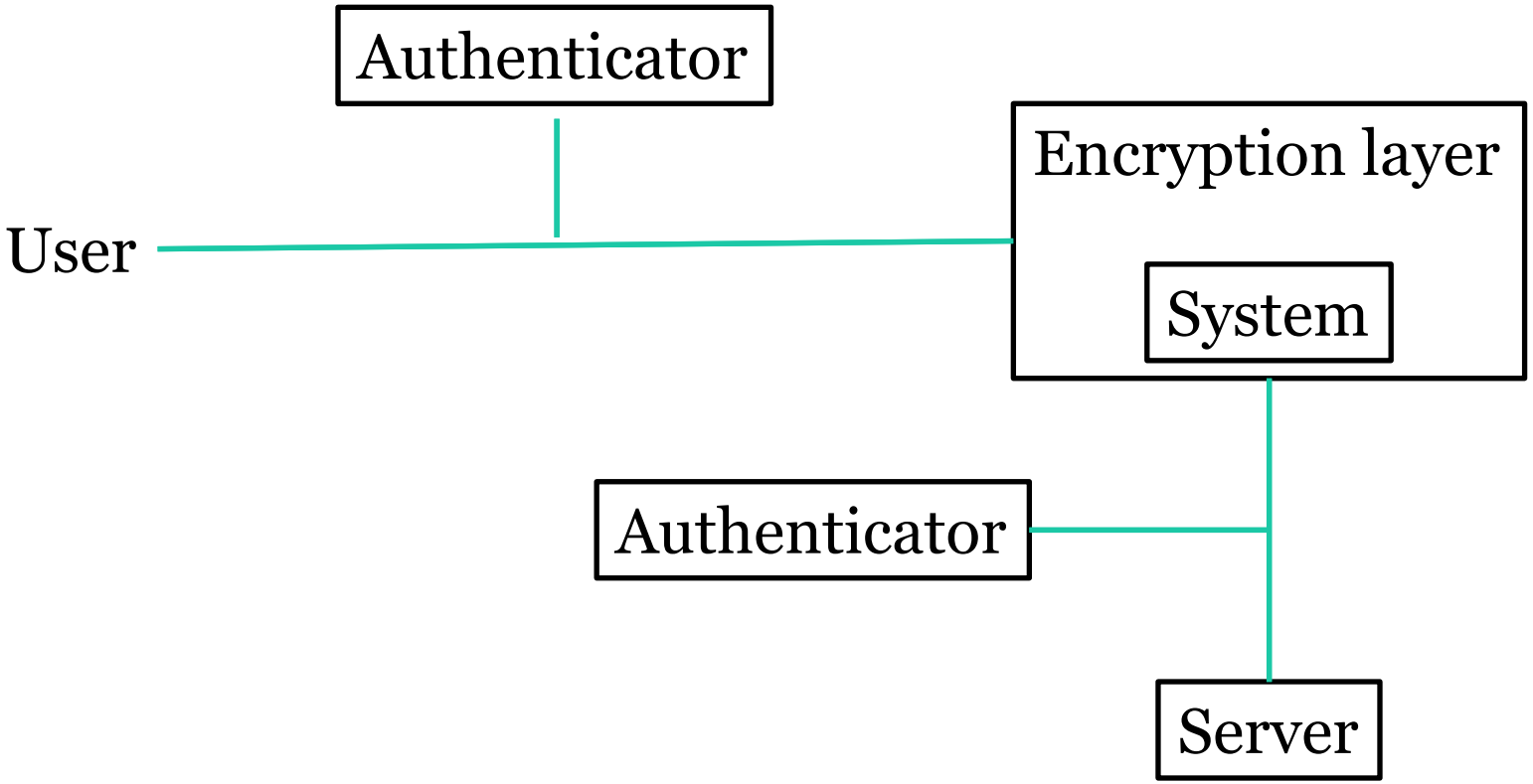
Availability

- Right information is available at the right time
- Important for everyone

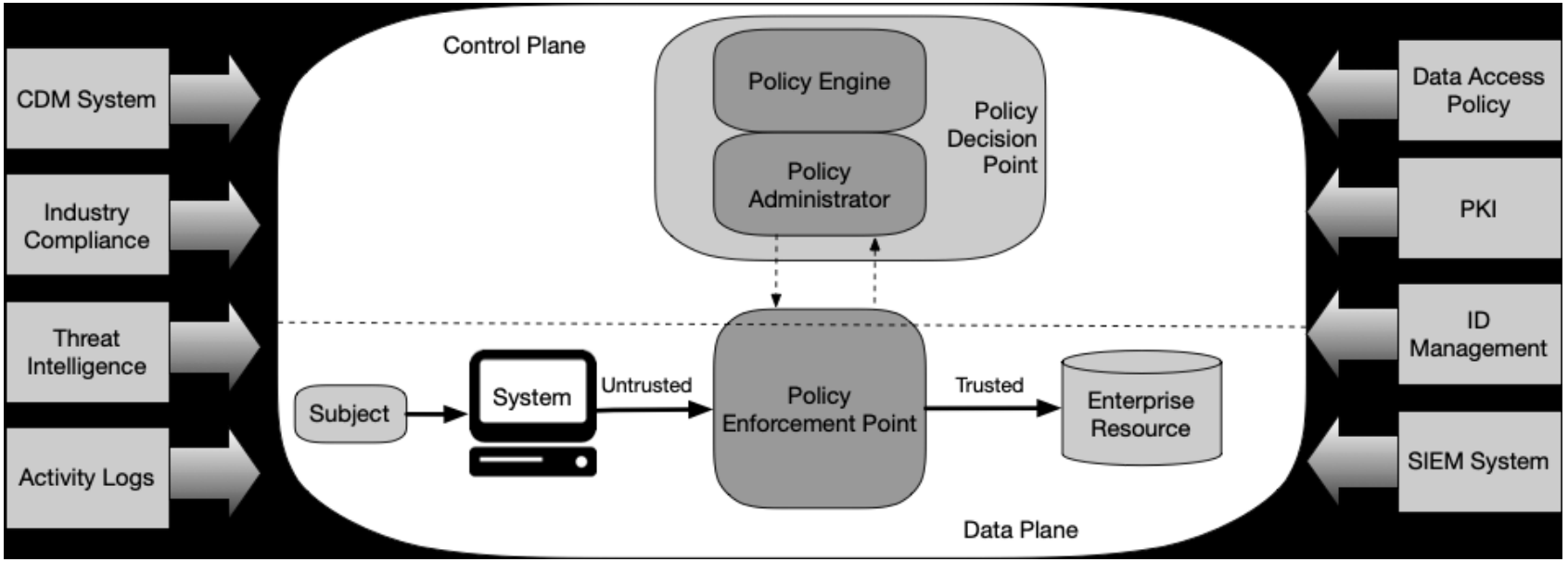
Integrity

- Only authorized users can modify, edit or delete data.
- E.g. bank systems

Security


Authenticator
Encryption layer
User
System
Authenticator
Server

Less naïve NIST Zero Trust logical components



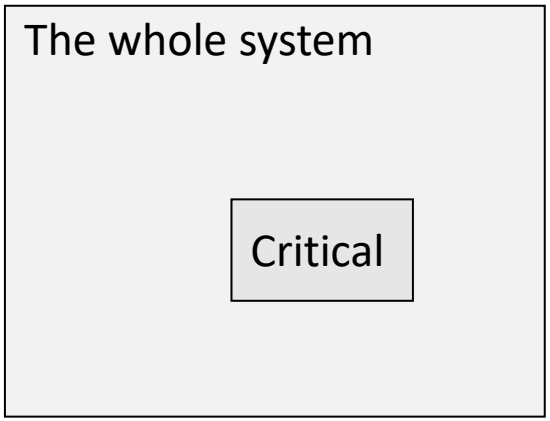
Rose, S. , Borchert, O. , Mitchell, S. and Connelly, S. (2020), Zero Trust Architecture, Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, [online], https://doi.org/10.6028/NIST.SP.800-207, https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=930420 (Accessed September 2, 2022)

Safety - absence of critical faults

Critical failures can create great damage to property, environment and lives.

E.g. military products

Isolate the most critical parts

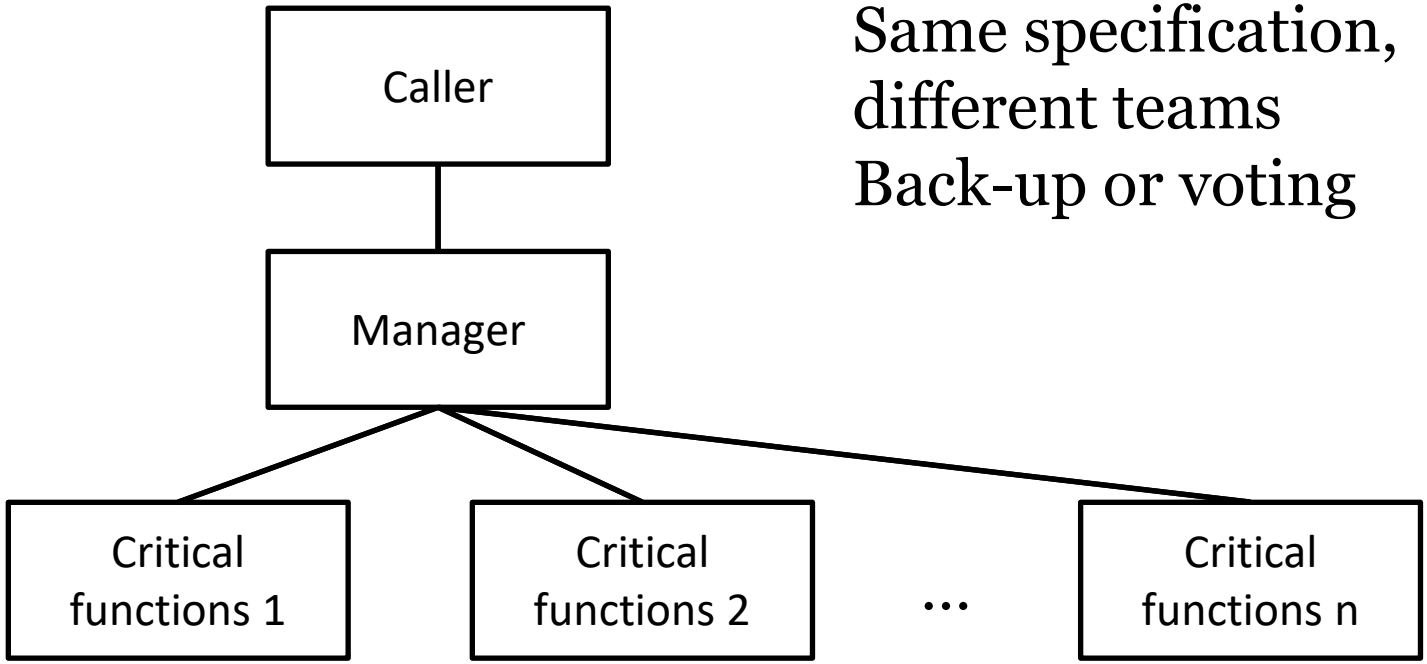


Design so that all safety critical operations are located in one or few modules / subsystems.

How can we validate that a safety critical system is correct?

- Formal validation?
- Testing?
- Software reviews?
- Experience?

Redundancy + Diversity



Same specification, different teams
Back-up or voting

Maximizing non-functional system characteristics with architectural design

Performance:

- Scale-up: Creating a small number of large subsystems,
- Scale-out: Parallel computations (see cloud)

Security:

- Maximized by layering systems with critical assets protected in the innermost layer
- No information up-read / down-flow

Safety:

- Maximized by placing critical safety functions in a small number of subsystems

Maximizing non-functional system characteristics with architectural design

Availability:

- Maximized through redundant subsystems to allow hot-swapping for updates

Maintainability:

- Maximized by creating a large number of small, independent subsystems

Balancing tradeoffs in architectural design

Performance:

- Maximized by creating a **small number** of **large** subsystems

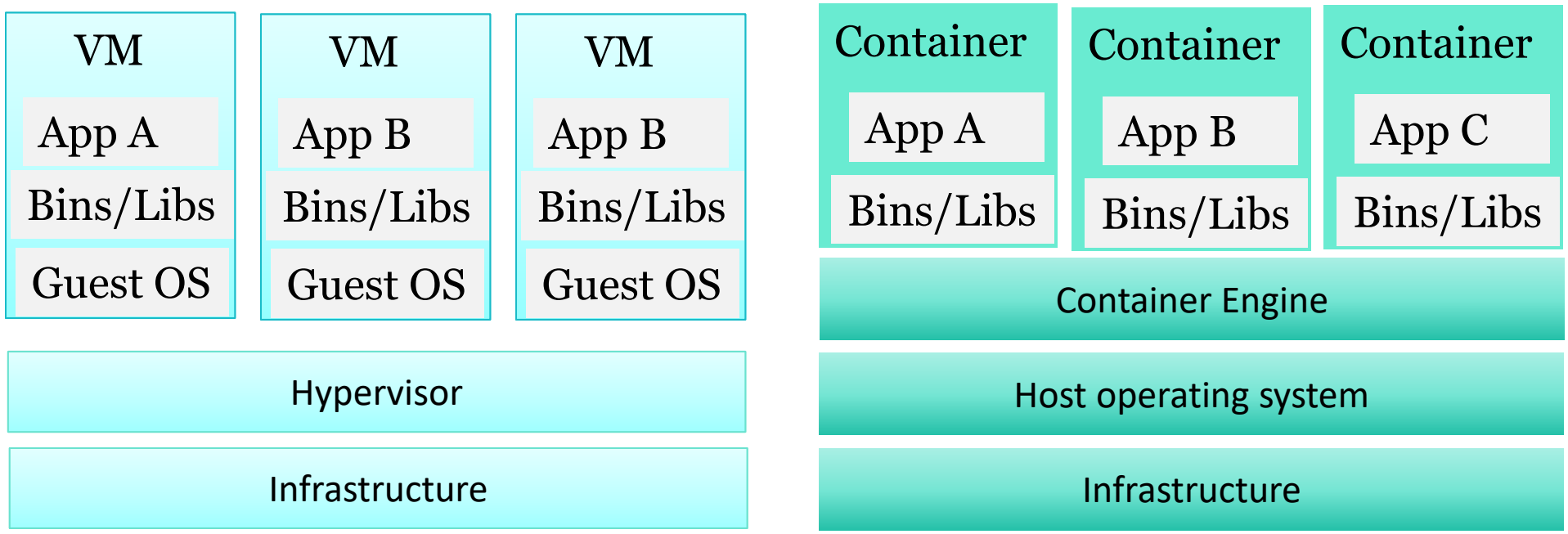
Maintainability:

- Maximized by creating a **large number** of **small**, independent subsystems

How can we create a portable system?

Historically, a major factor in technology decisions.

Containers and virtual machines



e.g. Virtual box, WMware

e.g. Docker

Reflections

Both adds computations => potential performance and resource penalties

VM:

+ run in isolation
+ interactive development
- takes time to build
- consumes storage

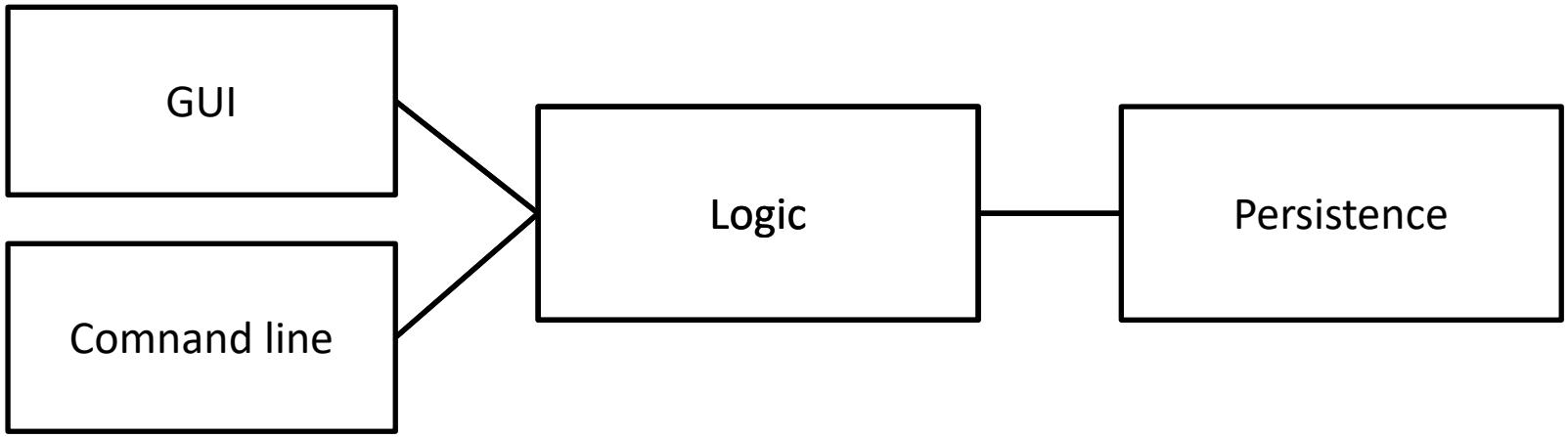
Containers:

+ faster iterations
+ ecosystem of software
- shared host exploits

Usability - How easy is it and what support exists to perform a task

Relevance
Efficiency
Attitude
Learnability

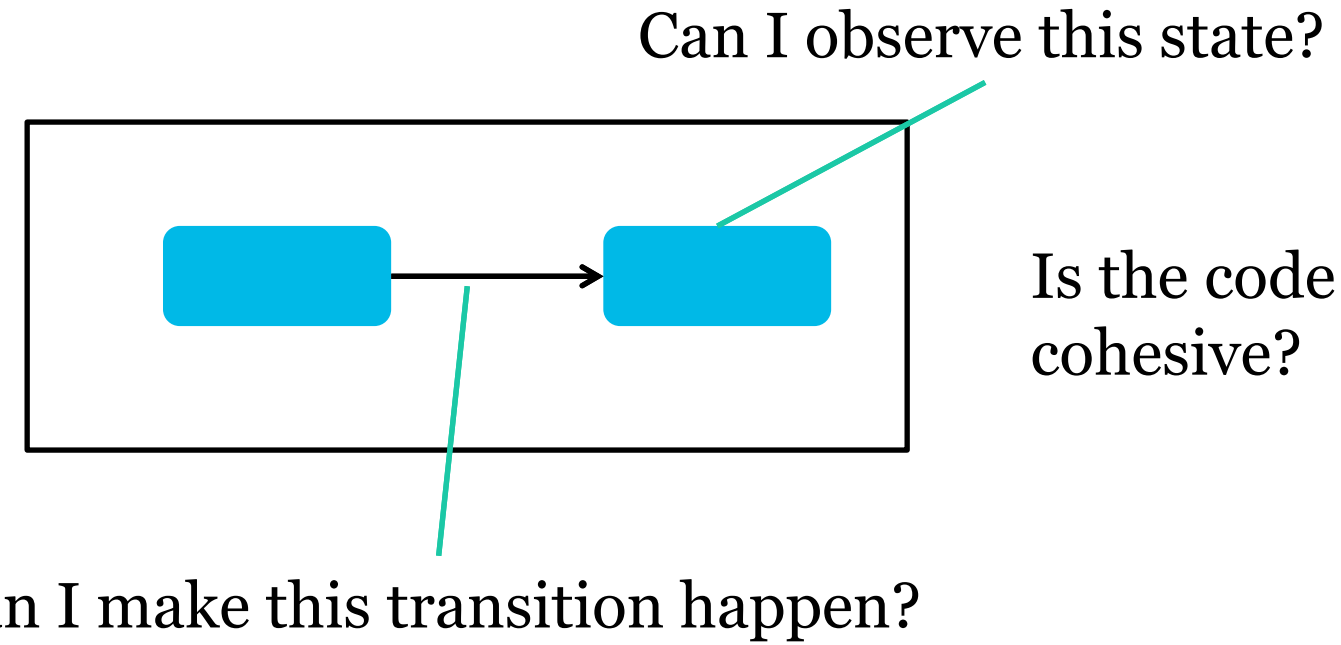
Separate interface and logic



How do we create a testable system?

At least 40% of the cost of well-engineered system is due to testing
(Bass et. al., 2003)

Control, observation, isolation



How can we create a system that is easy to maintain?

- Understandability
- Modifiability
- Testability

- Low coupling, high cohesion

Architectural Styles

Architectural patterns/styles:

- Are abstract descriptions of tried-and-tested solutions to common application problems
- Should describe when it is a good idea to use and when it should be avoided!

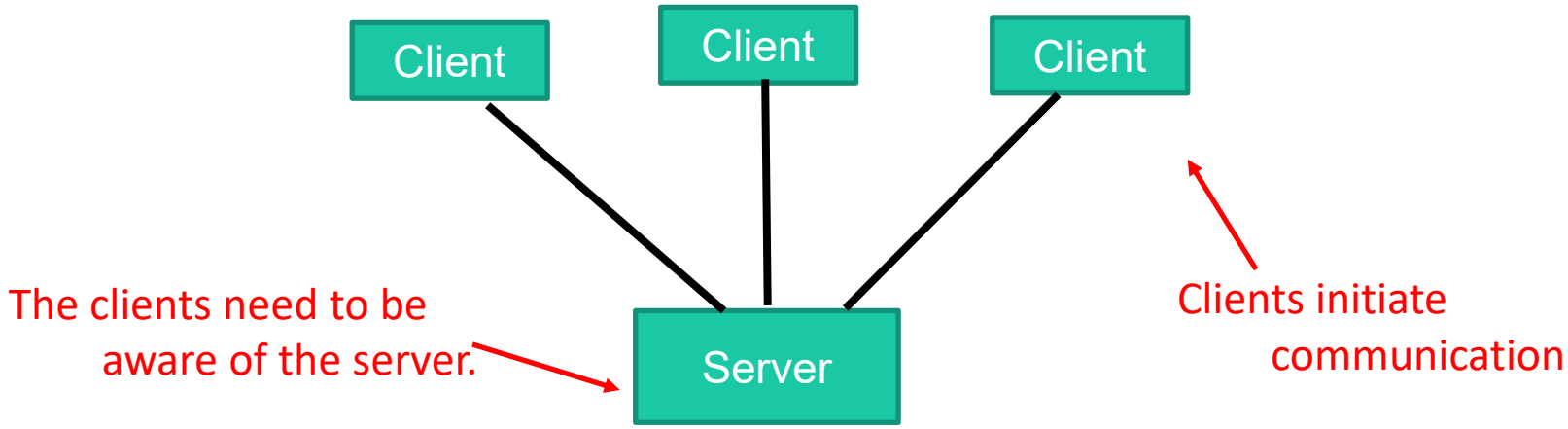
Architecture Styles / Patterns

Example of styles and patterns

- **Client-Server**
- **Layering**
- **Pipes-and-filters**
- **Service-oriented**
- **Model-View-Control (MVC)**
- **Repository**
- **Peer-to-Peer**

Discussed today

1. Client-Server



1. Client-Server

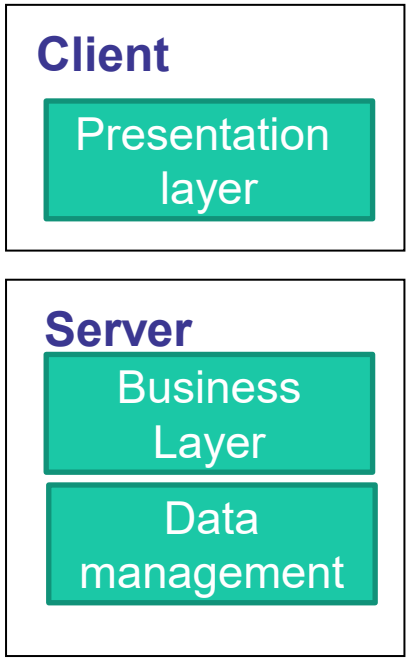
The clients need to be aware of the server.

Server

Clients initiate communication

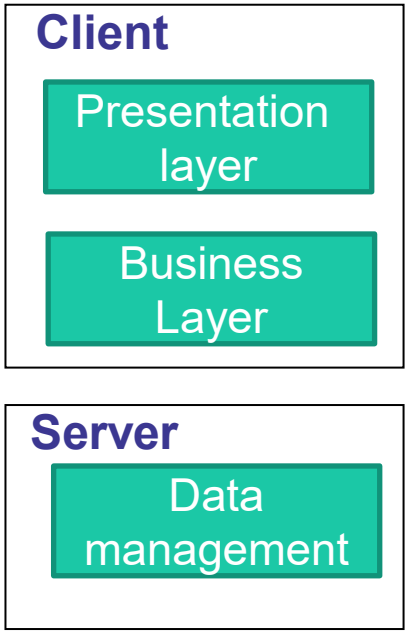
1. Client-Server

Two-Tier, Thin-client



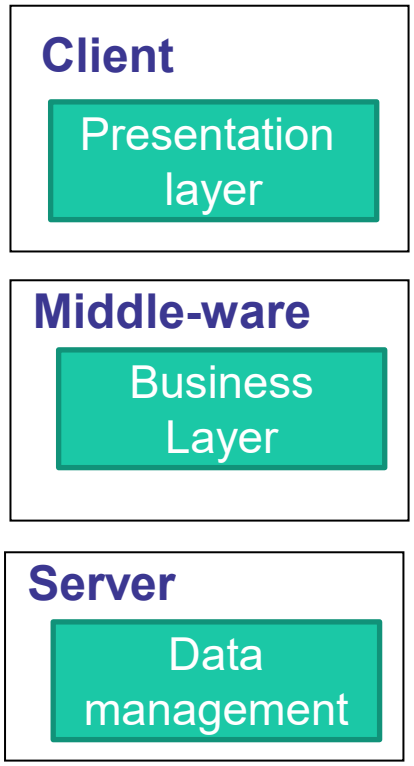
- Heavy load on server
- Significant network traffic

Two-Tier, Fat-client



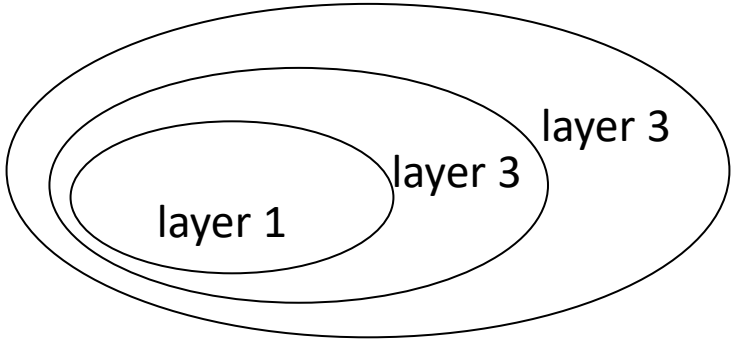
+ Distribute workload on clients
- System management problem, update software on clients

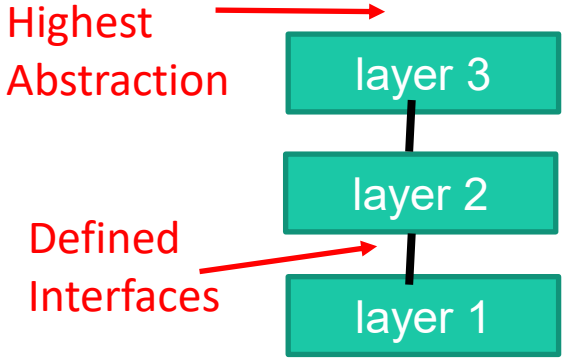
Three-Tier

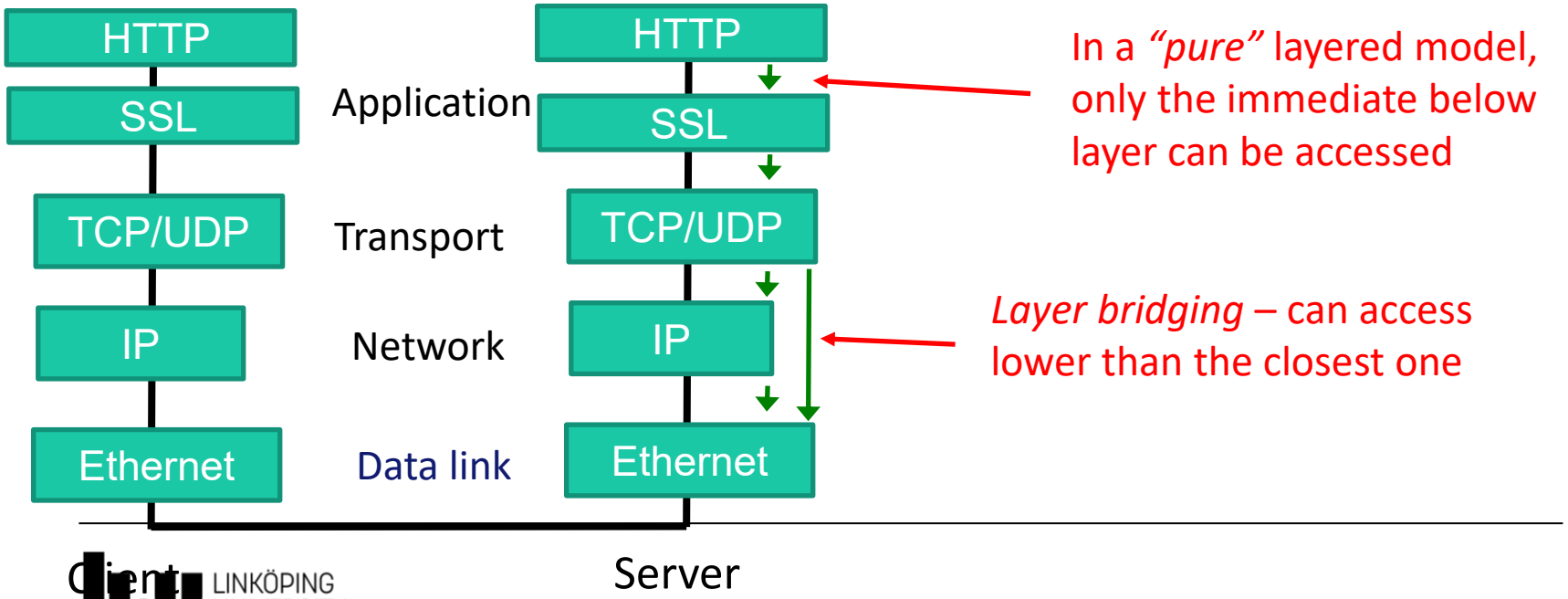


+ Map each layer on separate hardware
+ Possibility for load-balancing

2. Layers



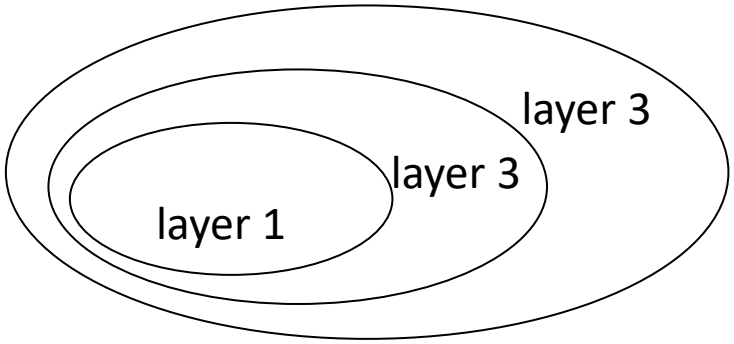




In a *"pure"* layered model, only the immediate below layer can be accessed

Layer bridging – can access lower than the closest one

2. Layers



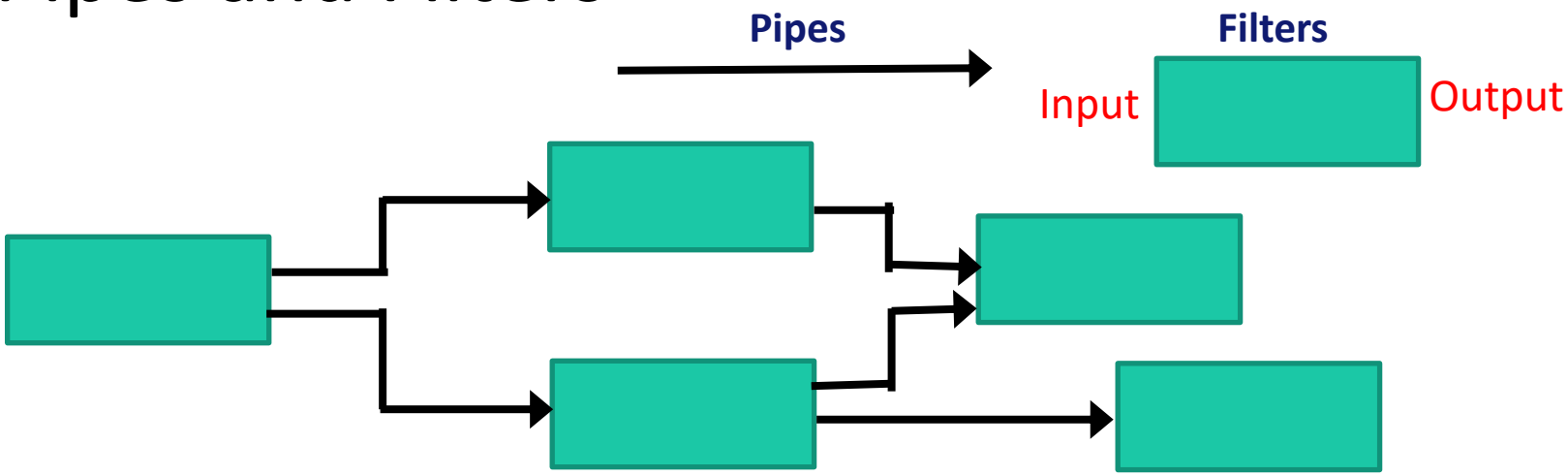
Pros

- Easy reuse of layers
- Support for standardization
- Dependencies are kept local - modification local to a layer
- Supports incremental development and testing

Cons

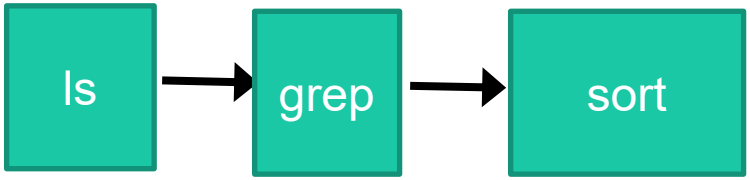
- Could give performance penalties
- Layer bridging loses modularity

3. Pipes and Filters

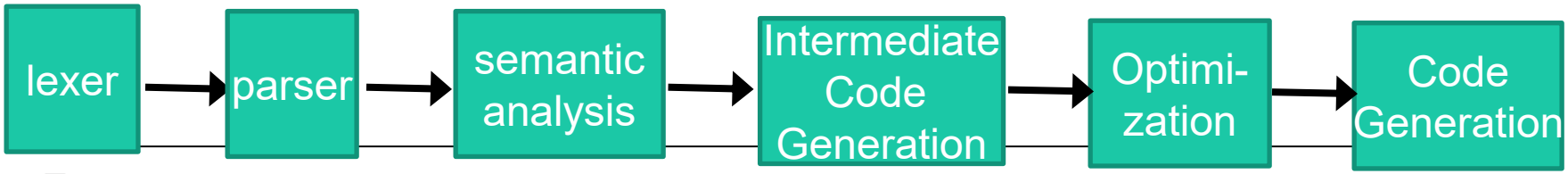


Example: UNIX Shell

```
ls -R |grep "html$" |sort
```



Example: A Compiler



Pipes and filters

Pros:

- Good understandability
- Supports reuse of filters
- Evolution eased
- Analyses of e.g. throughput are possible to early

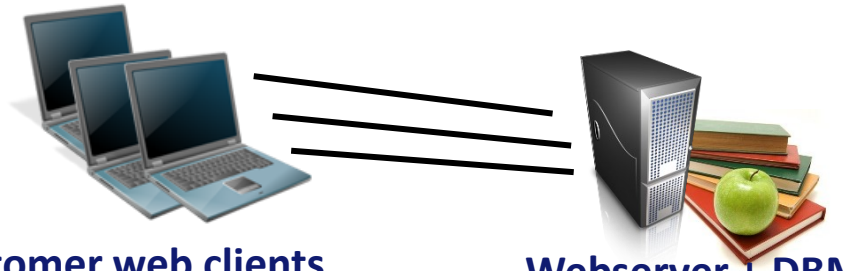
Cons:

- Redundant parsing of data => performance penalties

Case: (Service-Oriented Architecture)SOA and Amazon

Before 2001…

Two-tier architecture



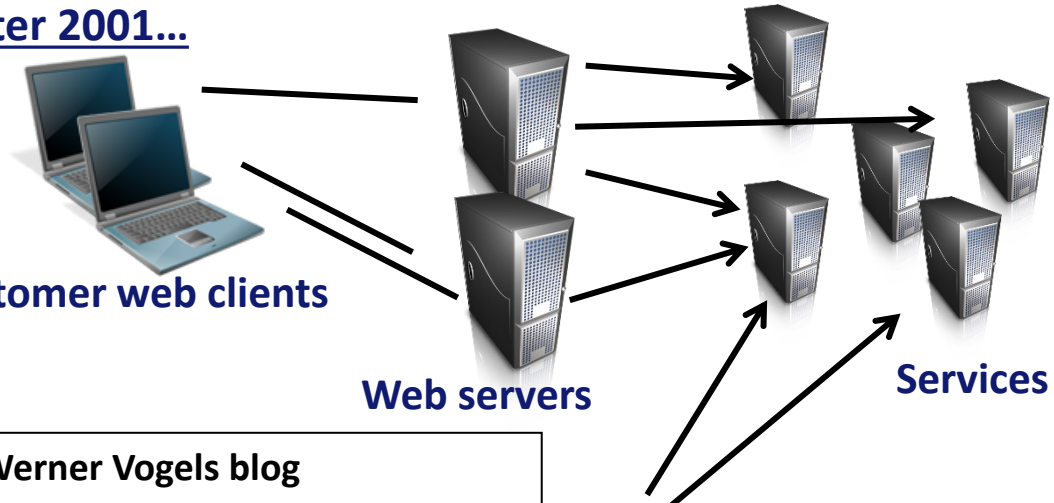
Customer web clients

Webserver + DBMS

Problems

- Scaling the DBMS
- Too complex software to maintain and develop

After 2001…



Customer web clients

Web servers

Services

Partner Companies (1 million)

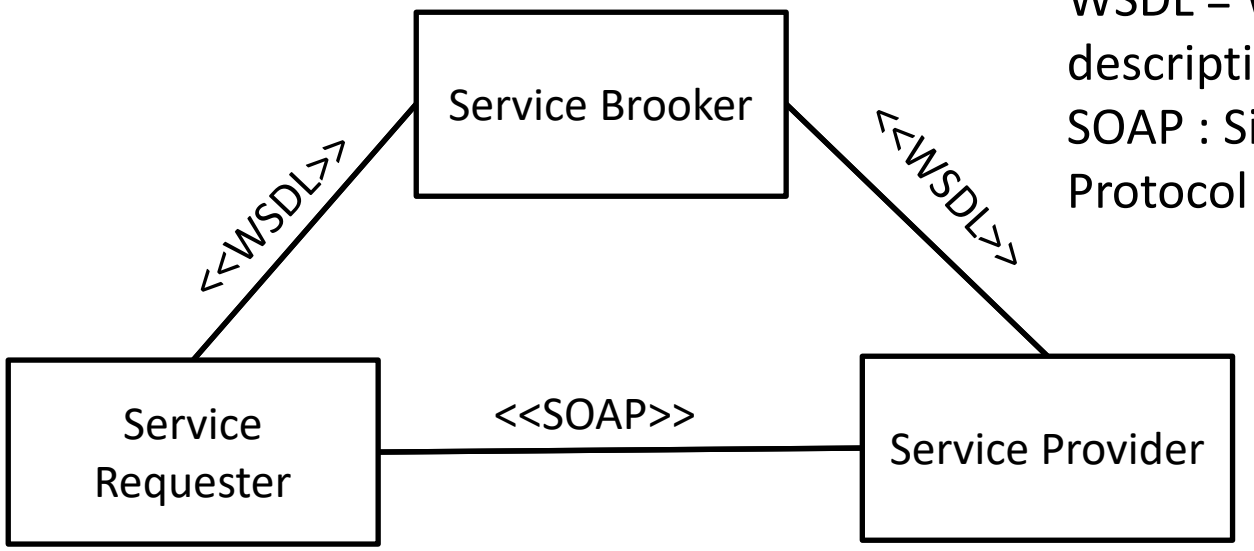
Key Success Factors

- Data encapsulated with business logic.
- No data sharing between services
- Independent dev teams for each service
- Developers have operational responsibility (you build, you run)

CTO Werner Vogels blog
www.allthingsdistributed.com

A bit more detail of Web services

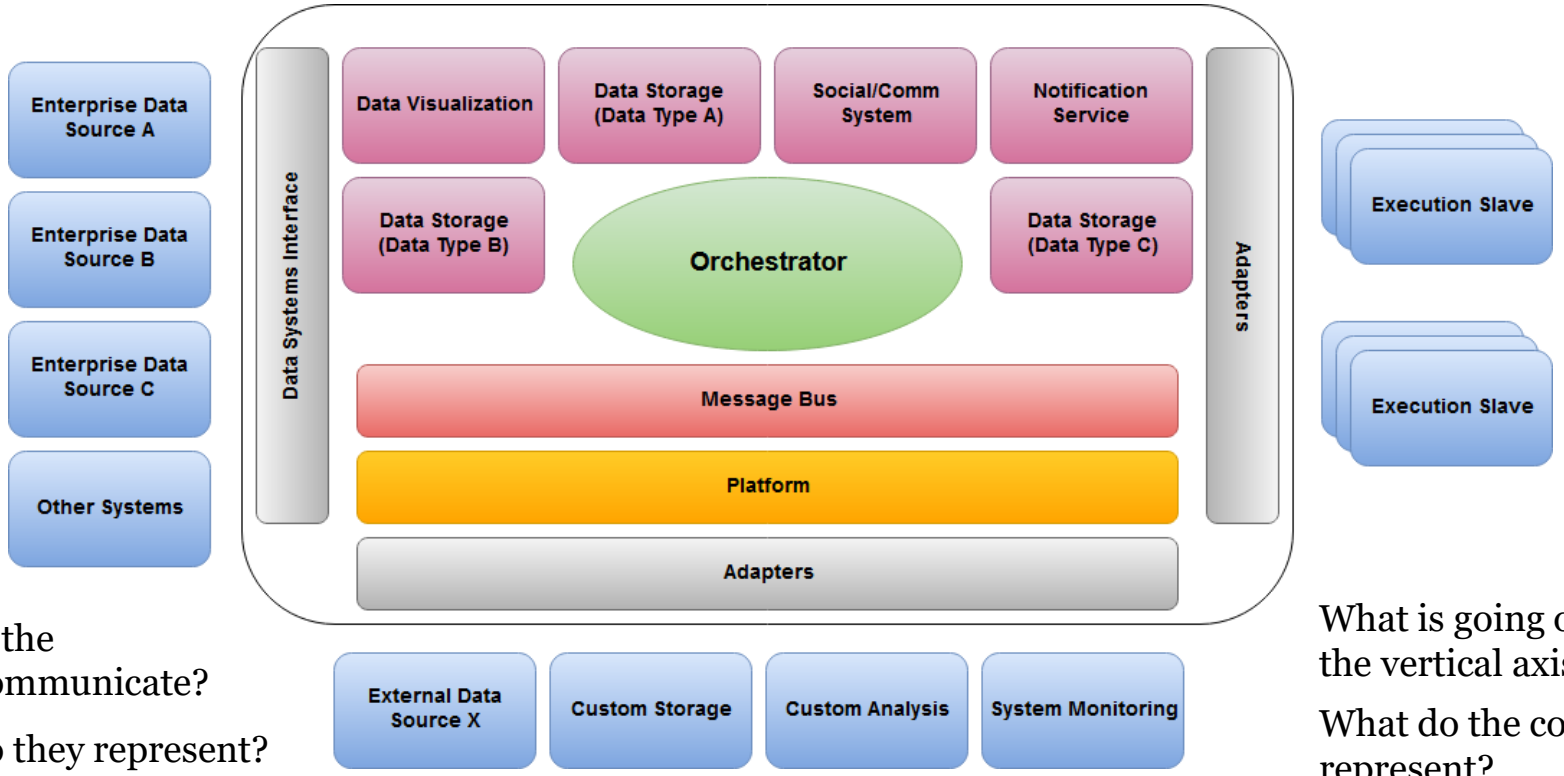
This is just an **example** of a SOA.



WSDL = Web service description language
SOAP : Simple Object Access Protocol

Documenting the Architecture

Adapted Example From Industry



How do the boxes communicate?

What do they represent?

Are their meanings consistent?

What is going on with the vertical axis?

What do the colors represent?

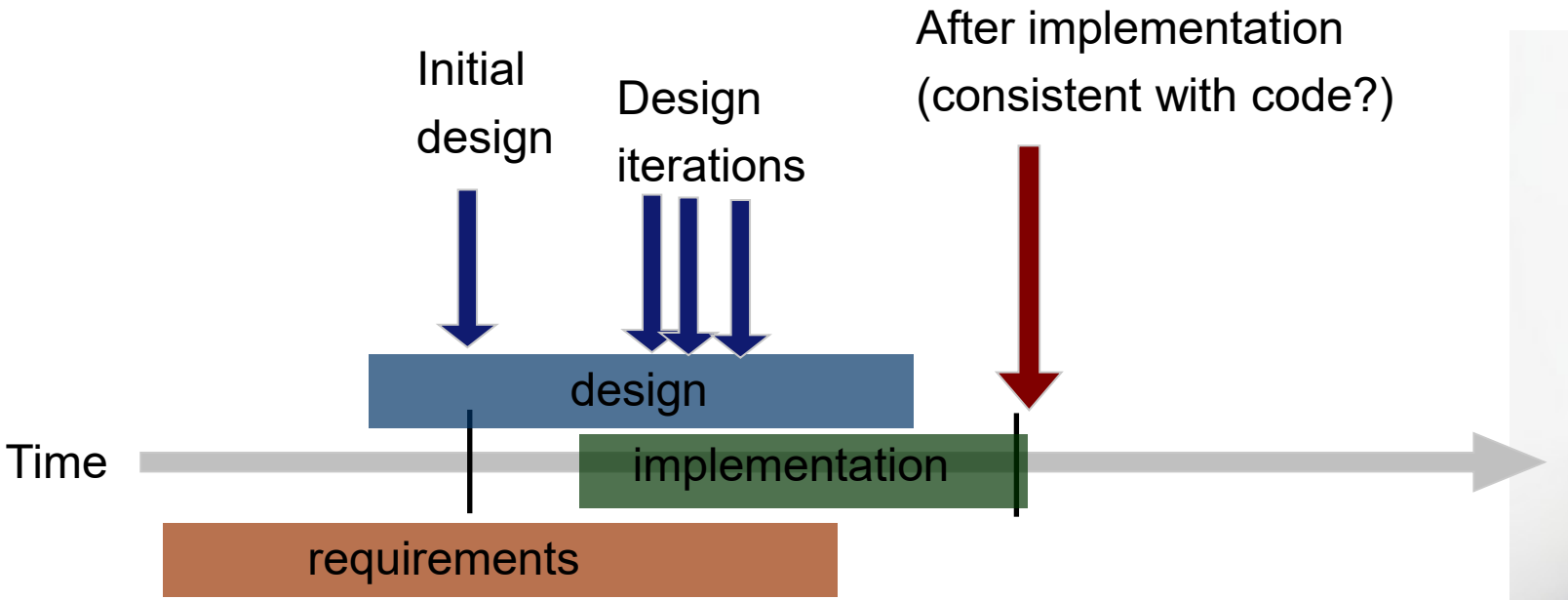
...?

Coming back to documents...

Write from the point of view of the readers...

Stakeholder	Use of the architect document
Requirements engineers	Negotiate and make tradeoffs among requirements
Architects/Designers	Resolve quality issues (e.g. performance, maintainability etc.)
Architects/Designers	A tool to structure and analyze the system
Designers	Design modules according to interfaces
Developers	Get better understanding of the general product
Testers and Integrators	Specify black-box behavior for system testing
Managers	Create teams that can work in parallel with e.g. different modules. Plan and allocate resources.
New software engineers	To get a quick view of what the system is doing
Quality assurance team	Make sure that implementation corresponds to architecture.

When to document?



The Architecture Notebook makes it easy to understand the architecture decisions

Maintains a list of:

- Issues
- Decisions
- Design patterns
- Pointer to code

- Supports iterative development of an architecture.
- Emphasizes the communication between roles
- Aligns with requirements.
- https://www.ida.liu.se/~TDDC88/openup/practice.tech.evolutionary_arch.base/workproducts/architecture_notebook_9BB92433.html?nodeId=9351a72b

Introduce the architecture and the document

1. Purpose

What will be included in the document?

2. Architectural goals and philosophy

What will drive the project?

E.g. High performance, adapt software, micro services

Critical issues addressed by the architecture

E.g. usability, scalability, modularity

3. Assumptions and dependencies

E.g. time, skills, resources, H/W dependencies

4. Architecturally significant requirements (ASR) determine the architecture

ASR can be:

- Important functions, e.g. persistence, authentication
- Non-functional, e.g. response time, portability
- High benefits to stakeholders, e.g. early demo wanted
- Handling a risk, e.g. availability of components

When the ASRs are met the architecture is stable!

5. List decisions together with constraints and justifications

Technology choices of all kinds

- E.g ”We will use a DBMS, since the user needs advanced search and filter.”
- E.g. ”We will use the React framework since the app will run in multiple browsers.”
- E.g. ”We will **not** use a service-oriented architecture since the customer don't think enough providers will register.”

6. Architectural Mechanisms are solutions that will be standardized in development

AMs evolve in different states, e.g.

Analysis mechanism	Design mechanism	Implementation mechanism
Persistence	RDBMS	MySQL
Communication	Message broker	RabbitMQ

Make design coherent

Support the buy/make decision

Architectural Mechanisms are often described in basic attributes

E.g. persistence:

- Granularity
- Volume
- Duration
- Retrieval mechanism
- Update frequency
- Survivability

7. Key abstractions are the most important concepts the system will handle

- Typically most high-level analysis classes, e.g. customer, catalogue, shopping-basket, payment
- Patterns, e.g. façade or observer
- Without key abstractions you cannot describe the system

8. Layers/architectural framework describe the components of an architectural style

- Elements of a box-and-line diagram, e.g. client and server
- Description of interfaces connecting elements

Summary

- Decompose-compose
- Coupling and cohesion
- Architectural views (implementation, execution, deployment)
- UML notations (Component, Subsystem, Artifact, Deployment)
- Quality factors vs architecture
- Architectural styles (Client-server, Layered, Pipes-and-filters, Service-oriented)
- The architectural notebook
- Much more in course: TDDE41 Software Architecture

Software Architecture /
Dániel Varró & Kristian Sandahl

www.liu.se

LINKÖPING UNIVERSITY