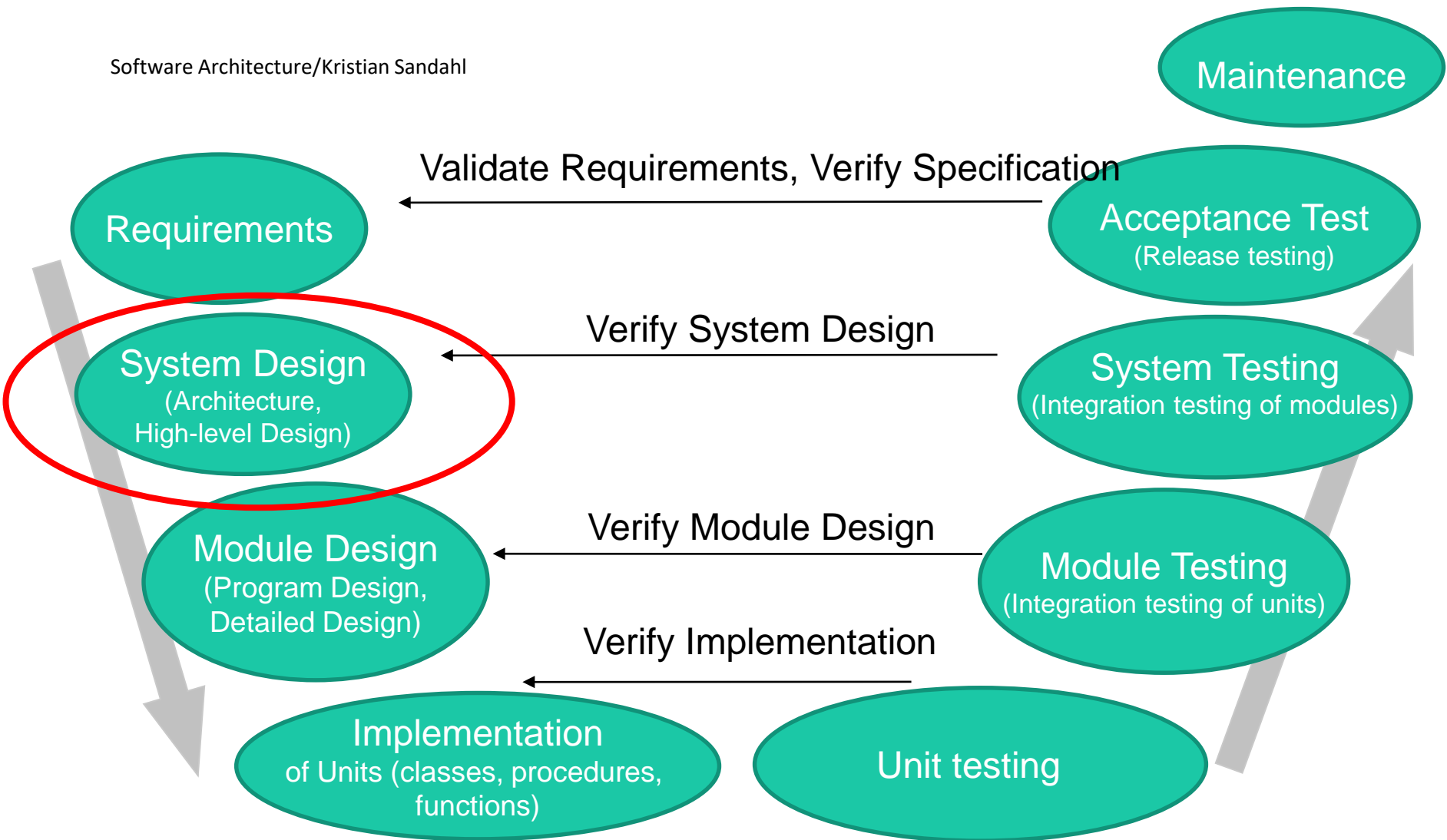


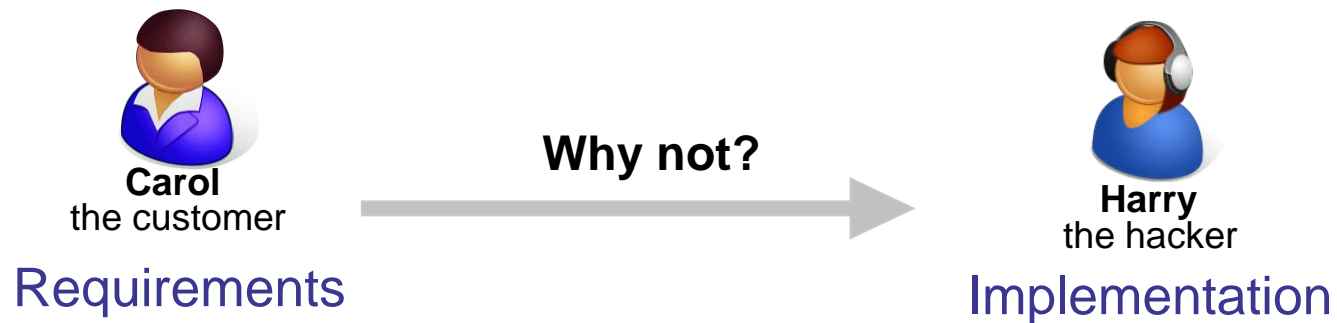
Software Architecture

Kristian Sandahl

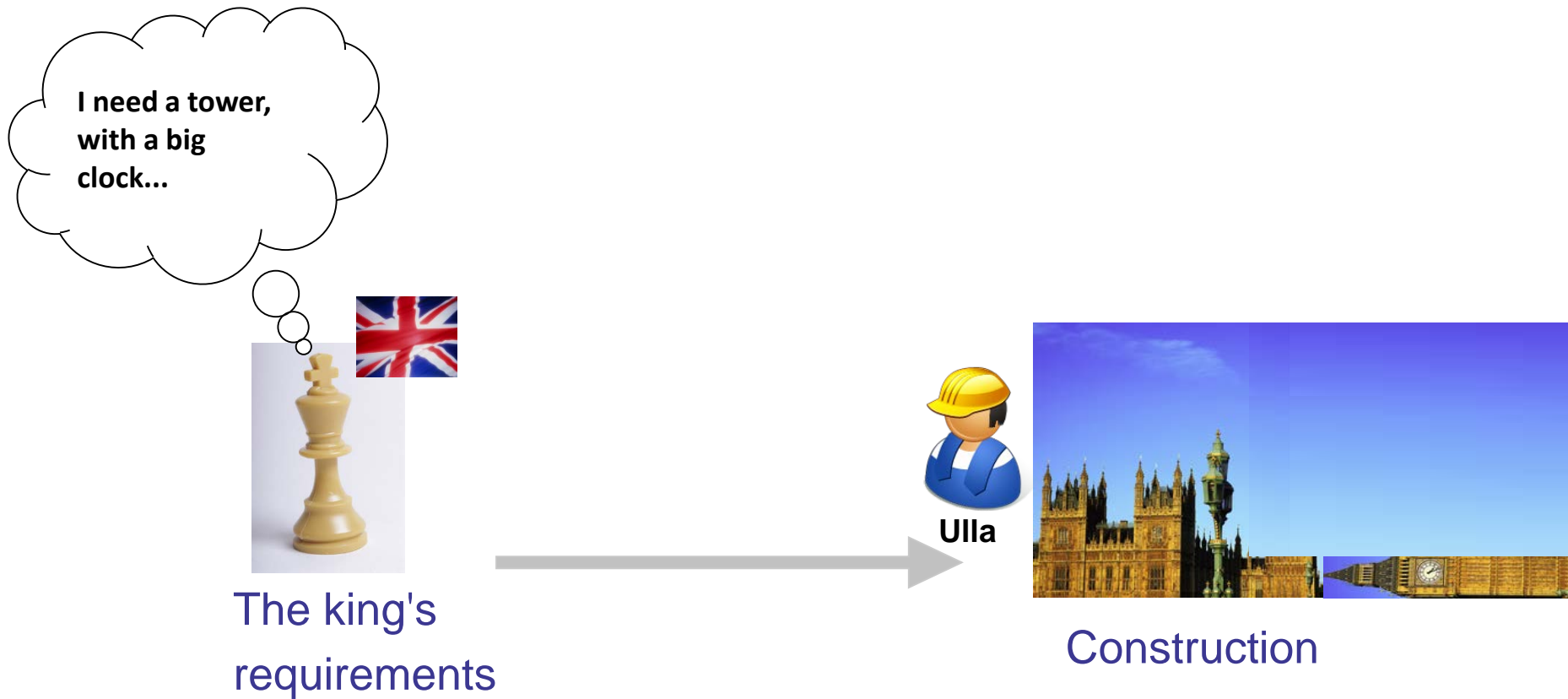


Project Management, Software Quality Assurance (SQA), Supporting Tools, Education

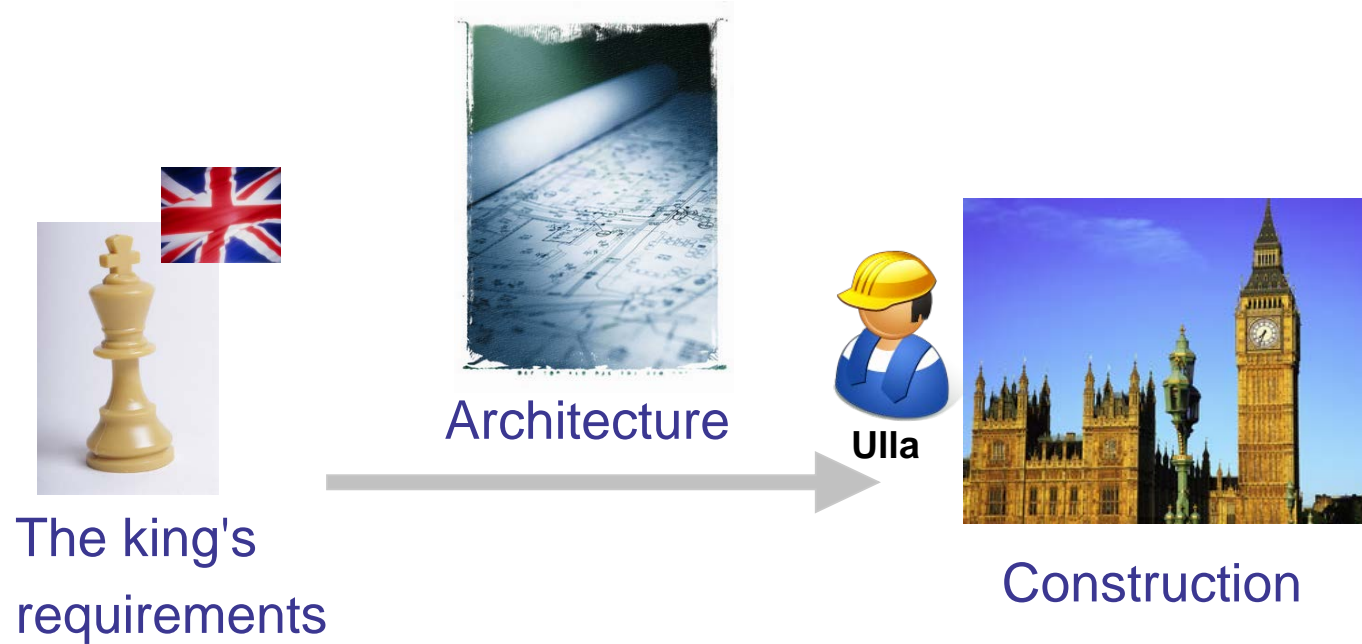
Why should we design a system?



Constructing a building...



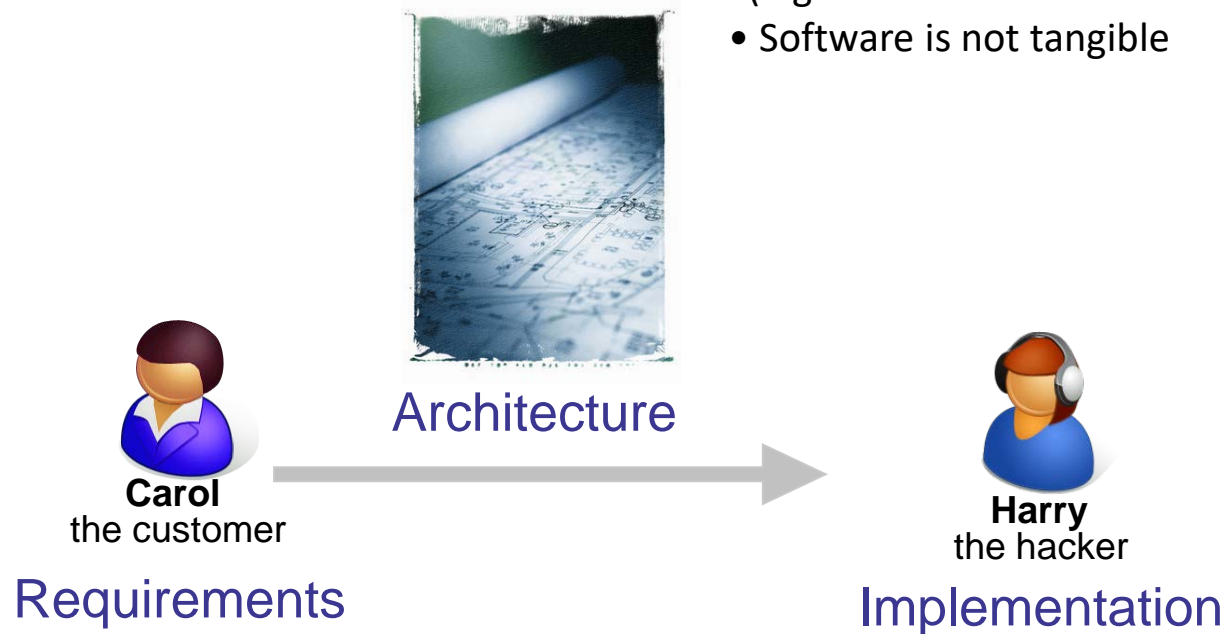
Constructing a building...



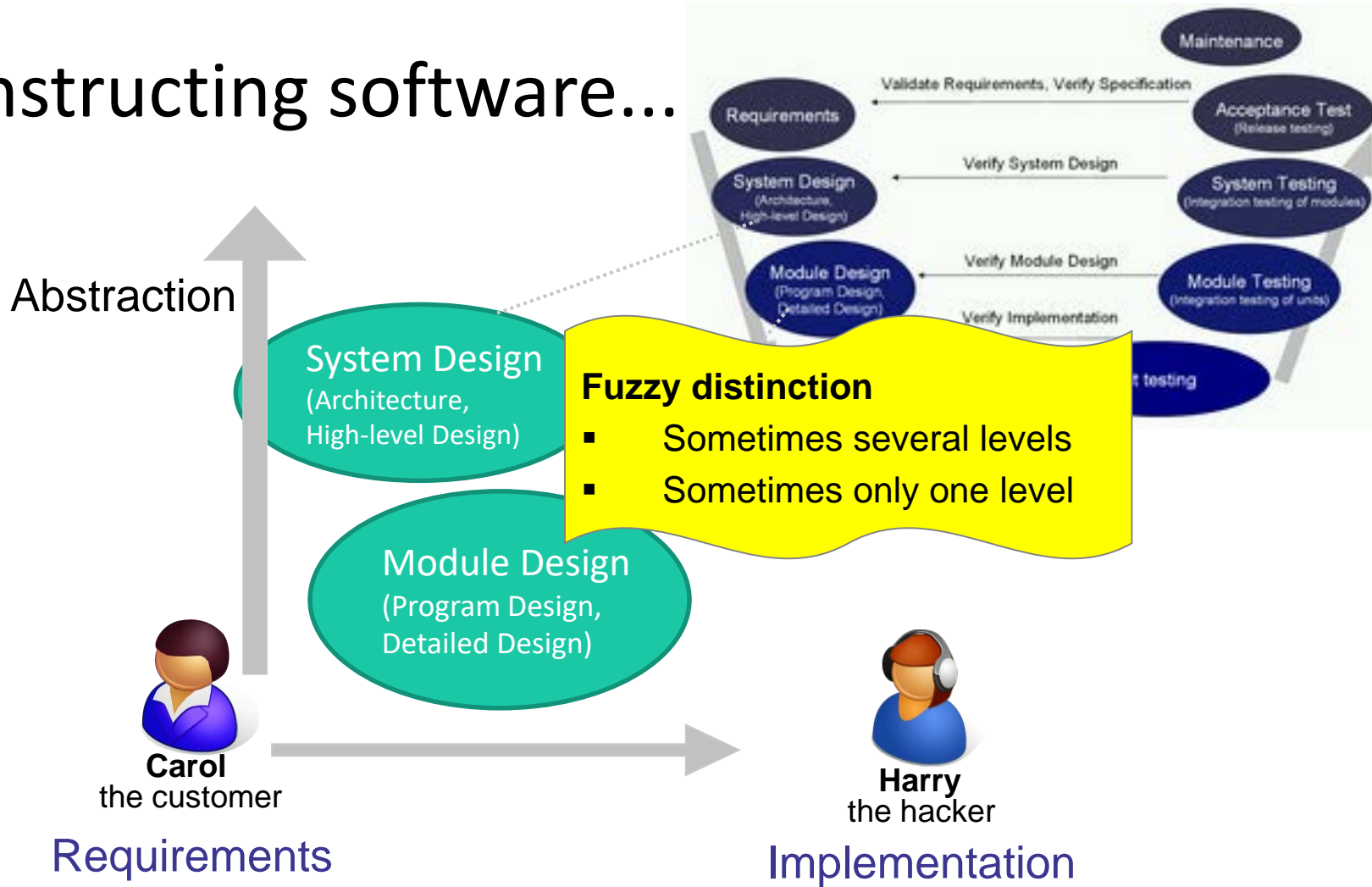
Constructing software...

Software is different

- No physical natural order of construction (e.g. start with the foundation of the house)
- Software is not tangible



Constructing software...



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.

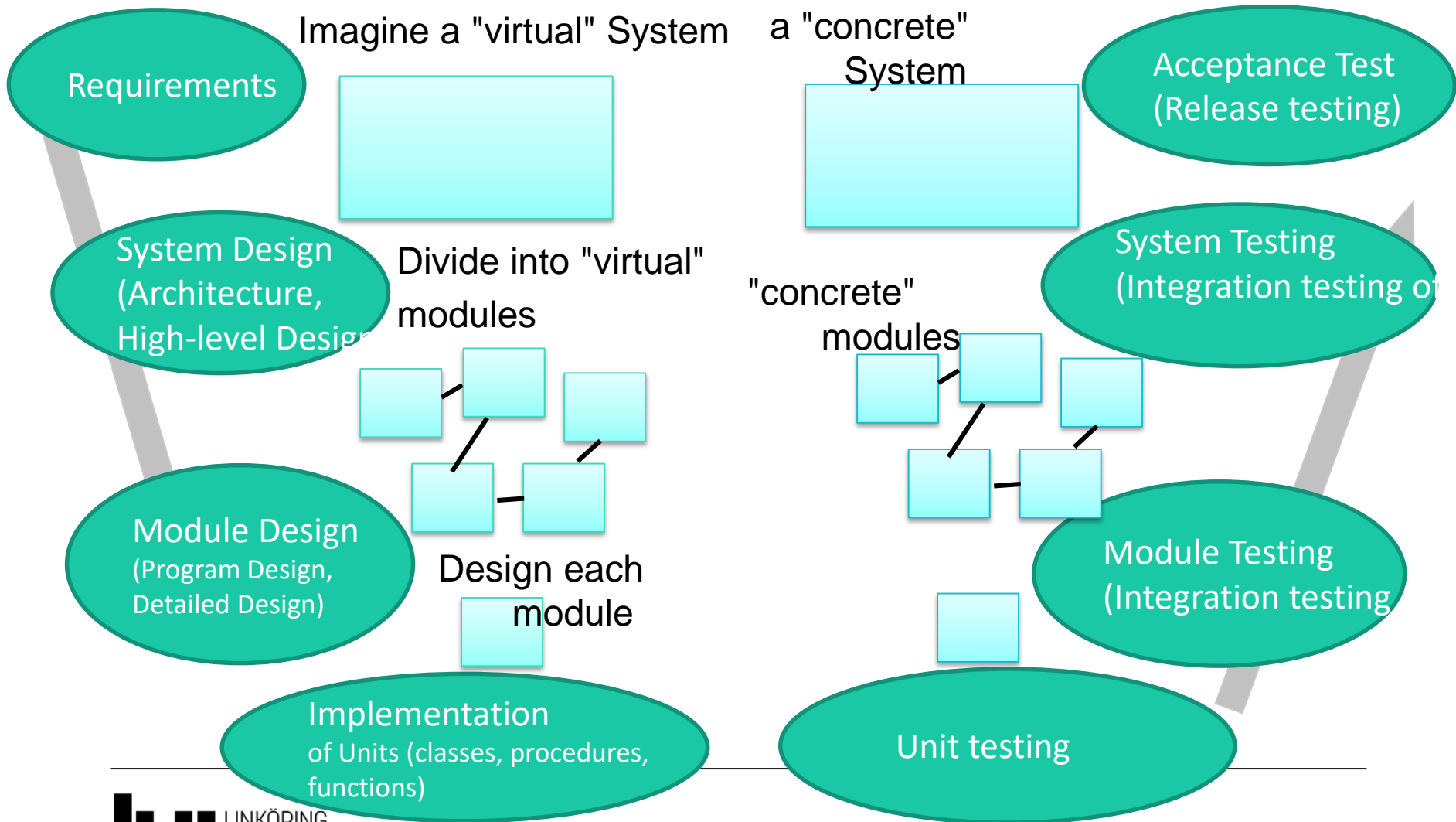


Large-scale reuse

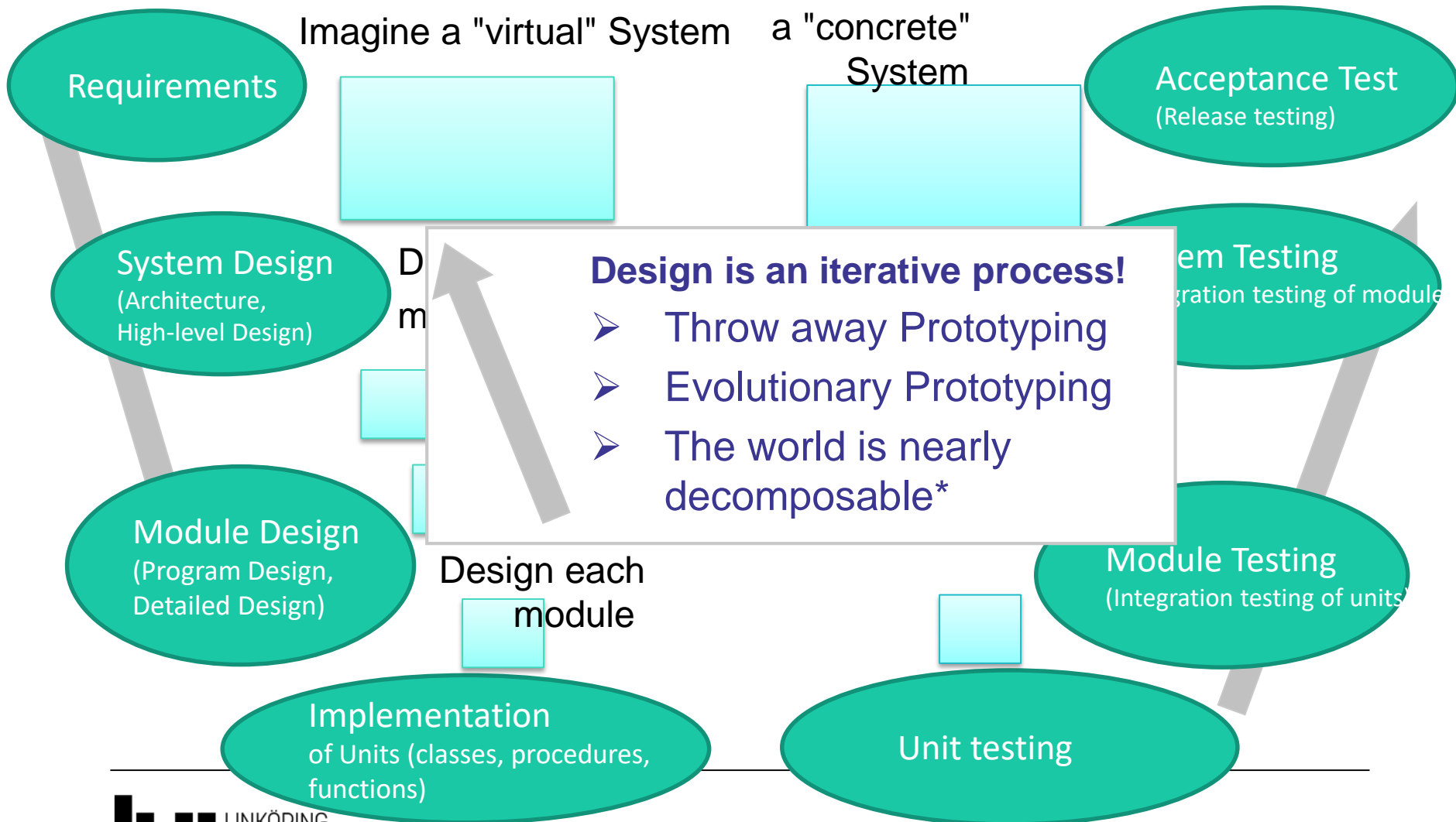
If similar system have common requirements, modules can be identified and reused.

(Bass et.al., 2003)

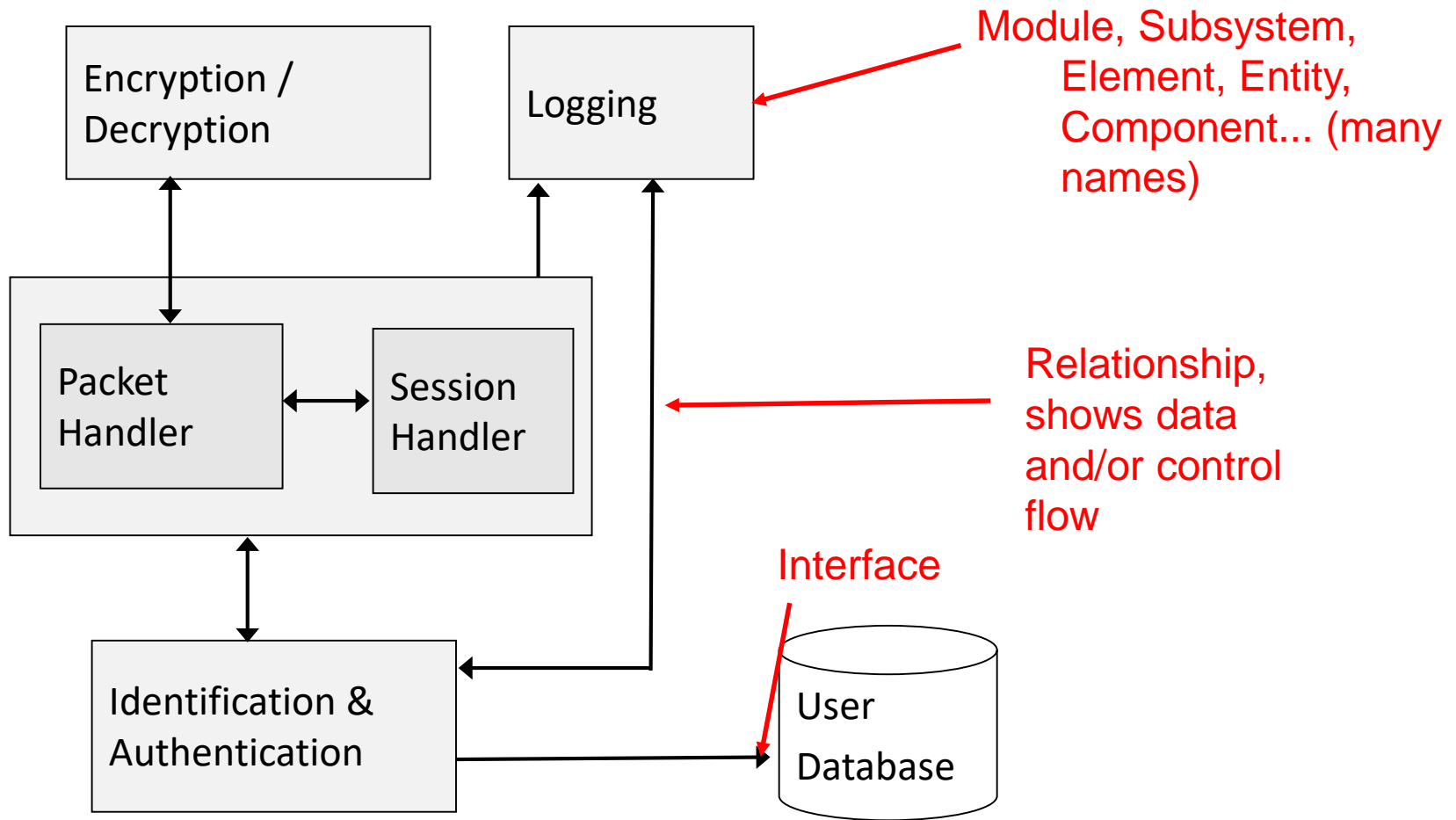
Analyze and Synthesis a system (decompose and compose)



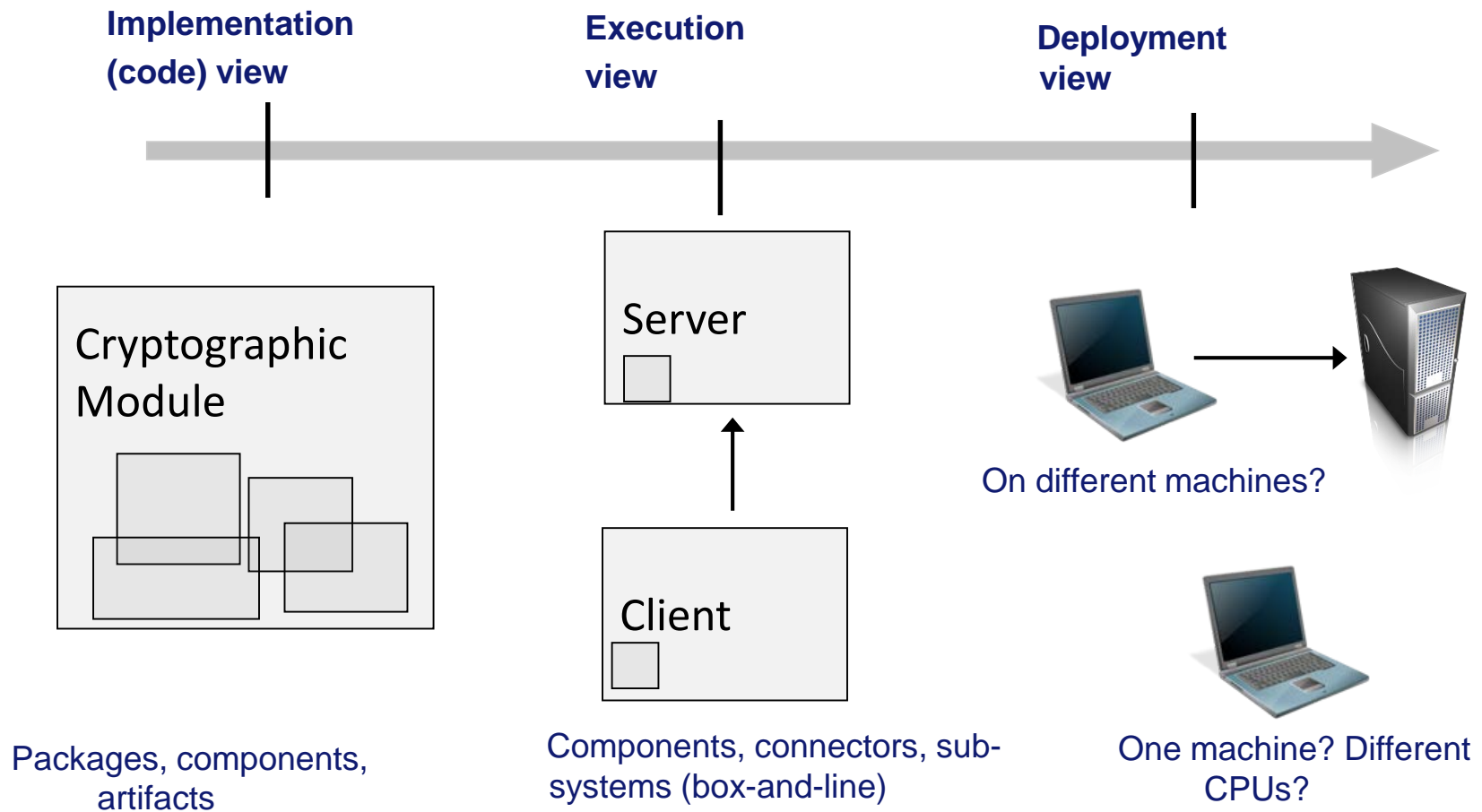
Analyze and Synthesis a system (decompose and compose)



Box-and-line diagrams...



Architectural views



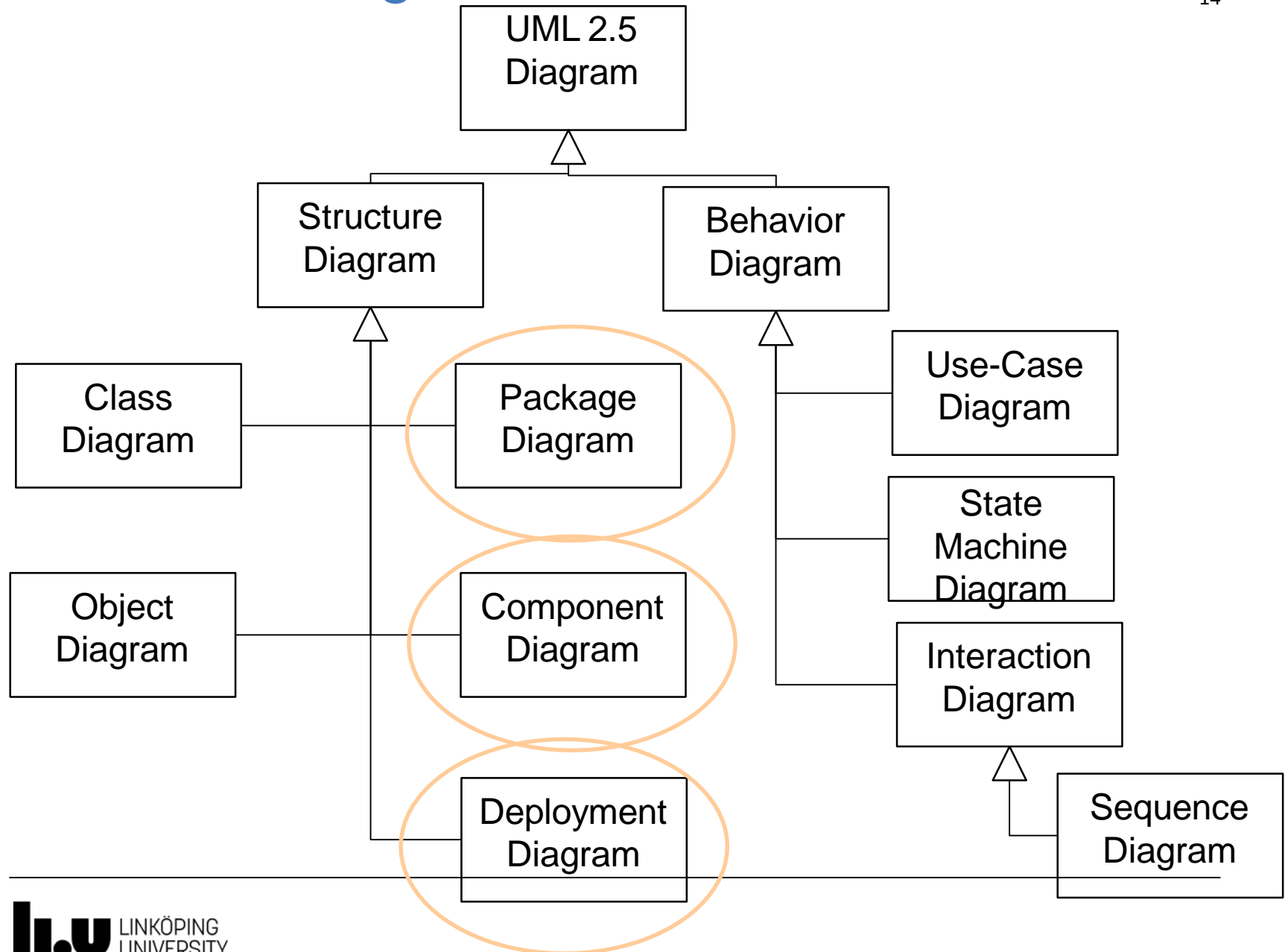
Unified Modeling Language

- Wide-spread standard of modeling software and systems
- Several diagrams and perspectives
- Often needs a text of assumptions and intentions
- Many tools tweak the standard



Well-known Diagrams of UML in architecture

14



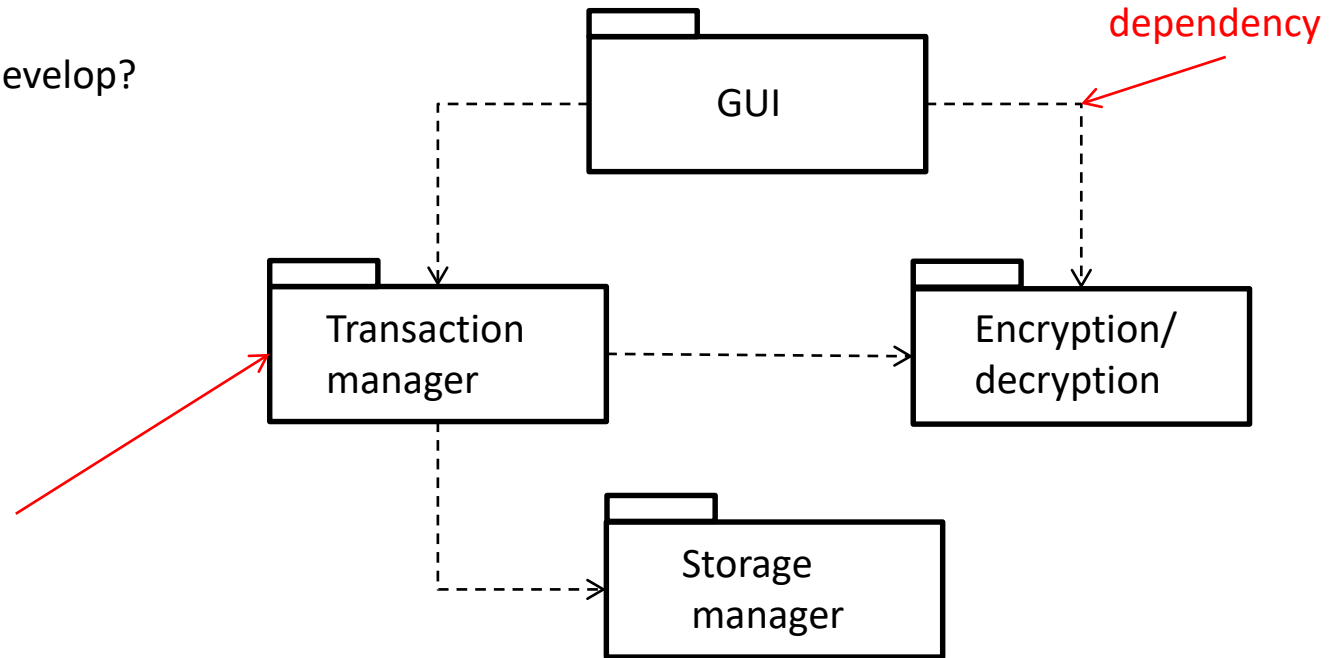
Implementation view with packages

A developer's perspective:

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

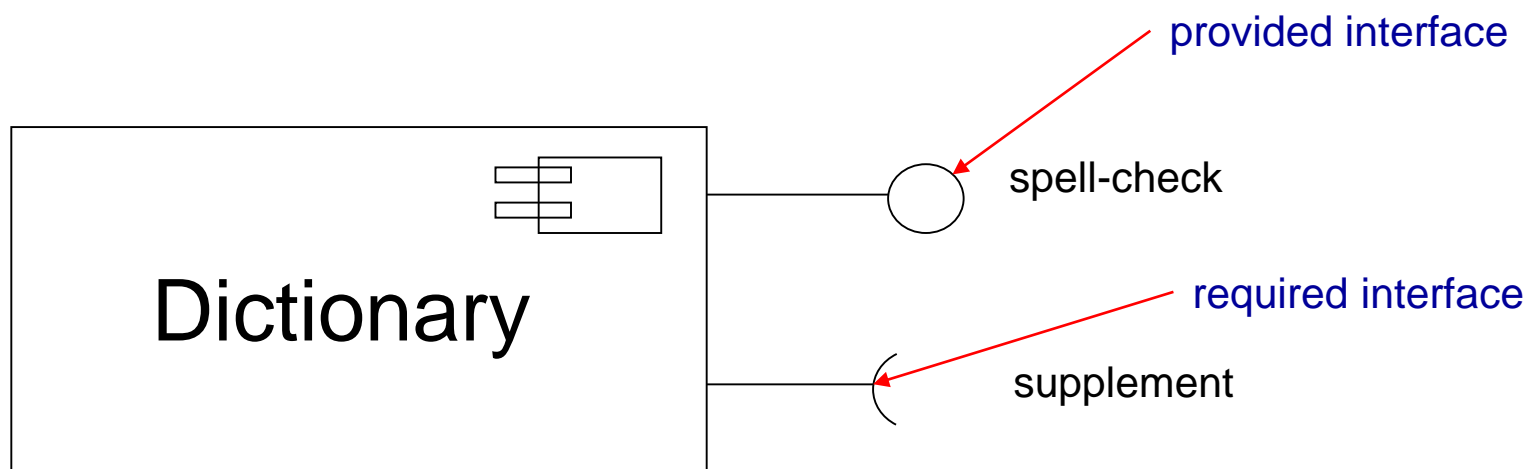
Package

- Organize work
- Compile together
- Name space

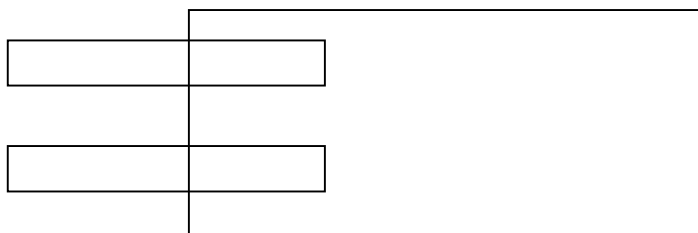


Packages can be used to give an overall structure to other things than code, eg. Use-cases and Classes

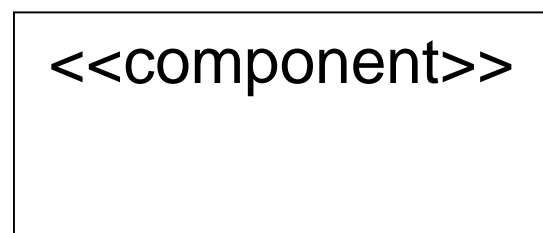
Component diagram with interfaces



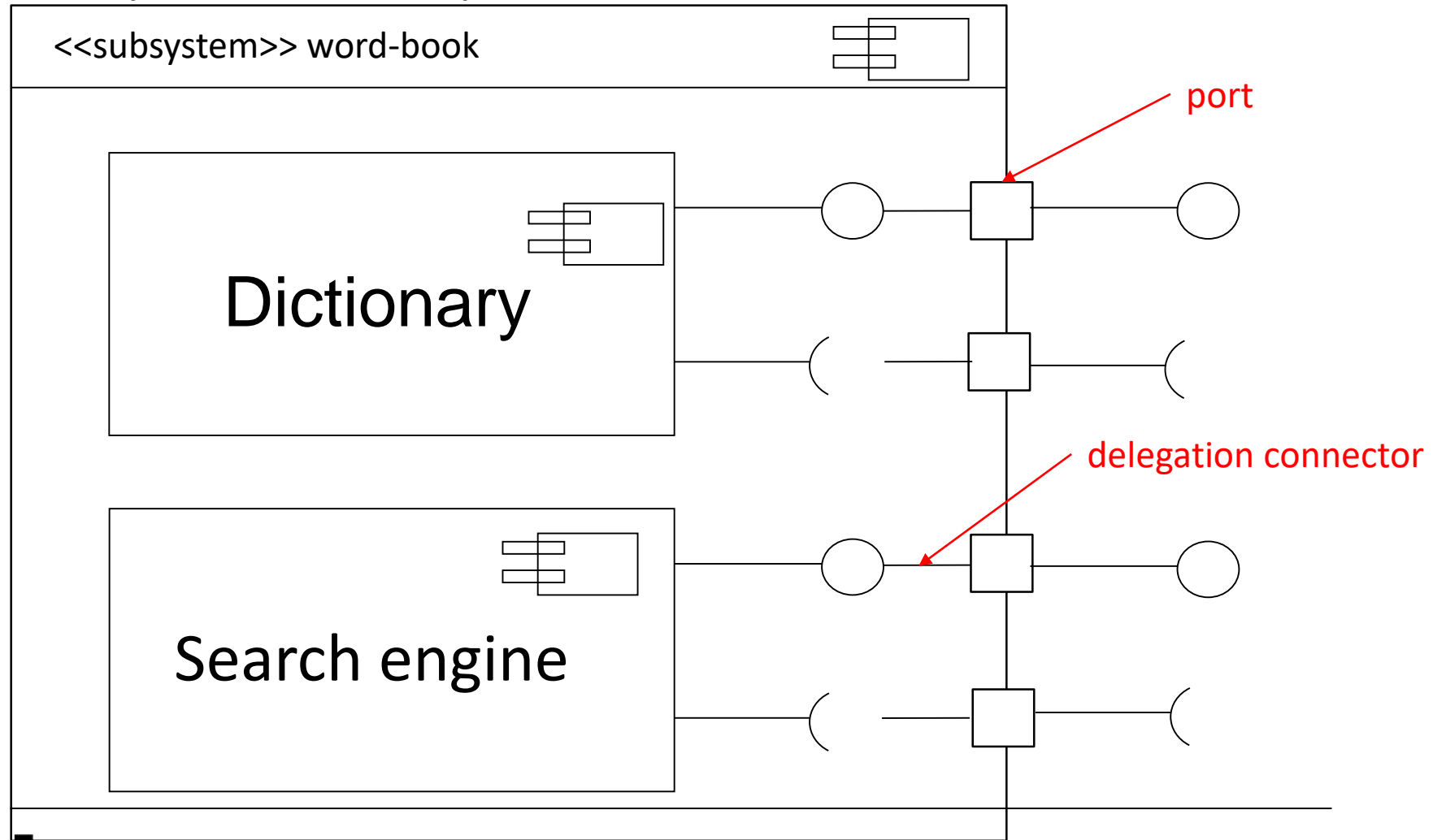
Older notation:



Alternative notation:

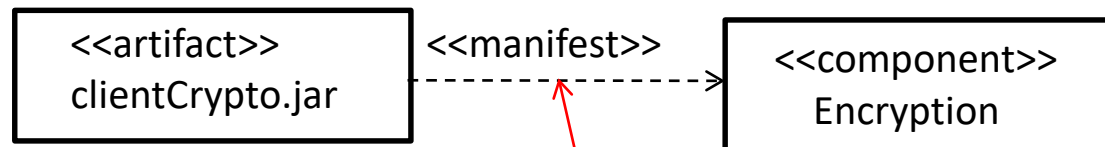
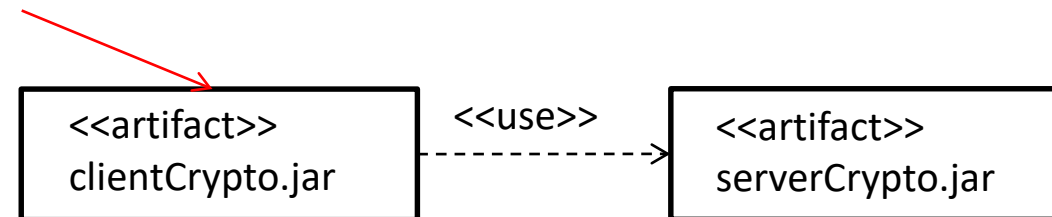


Subsystem with components



Artifacts

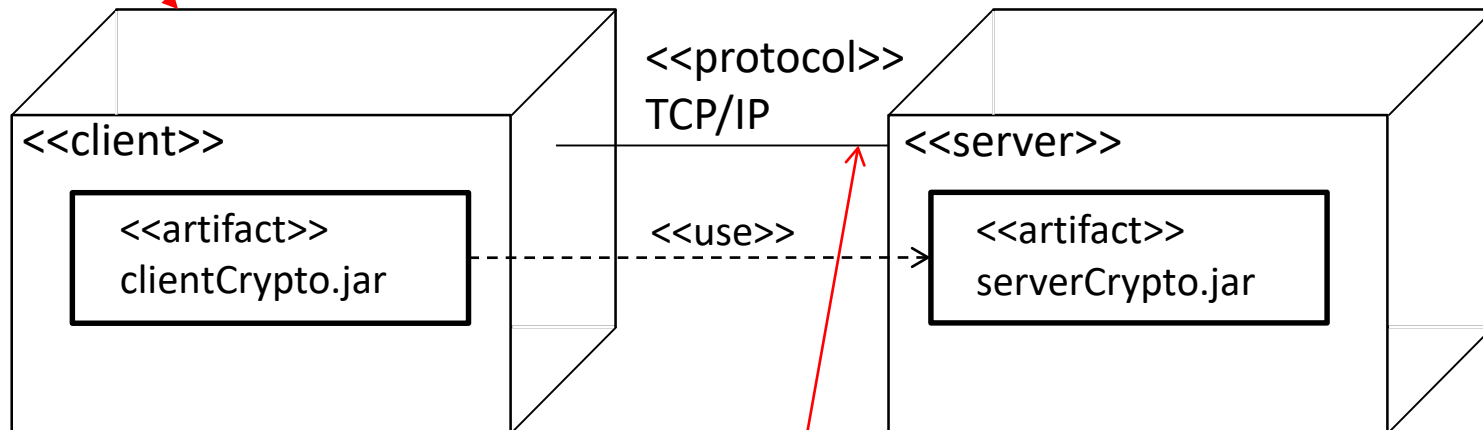
Physical code, file, or library



The artifact implements
the component

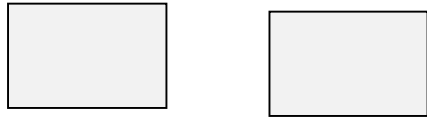
Deployment view in UML

Node, physical hardware

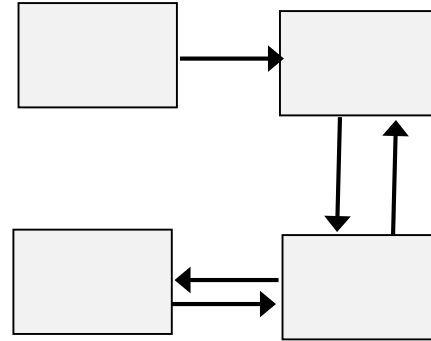


Communication path

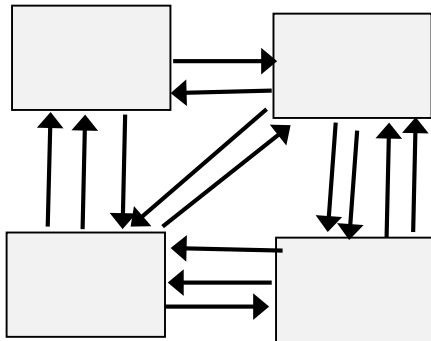
Coupling - dependency between modules



Uncoupled - no dependences



Loosely coupled - few dependencies



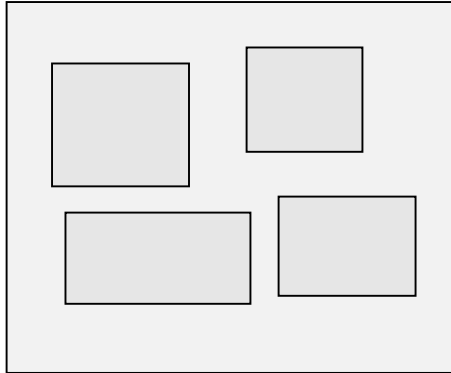
Highly coupled - many dependencies

What do we want?

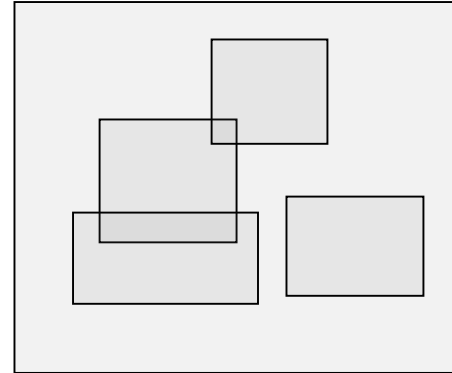
Low coupling. Why?

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

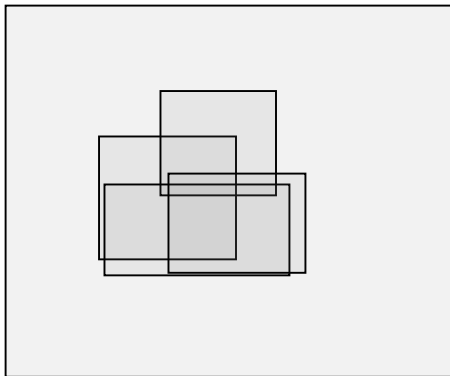
Cohesion - relation between internal parts of the module



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



Medium cohesion - some logically related function. E.g. IO related functions.



High cohesion - does only what it is designed for

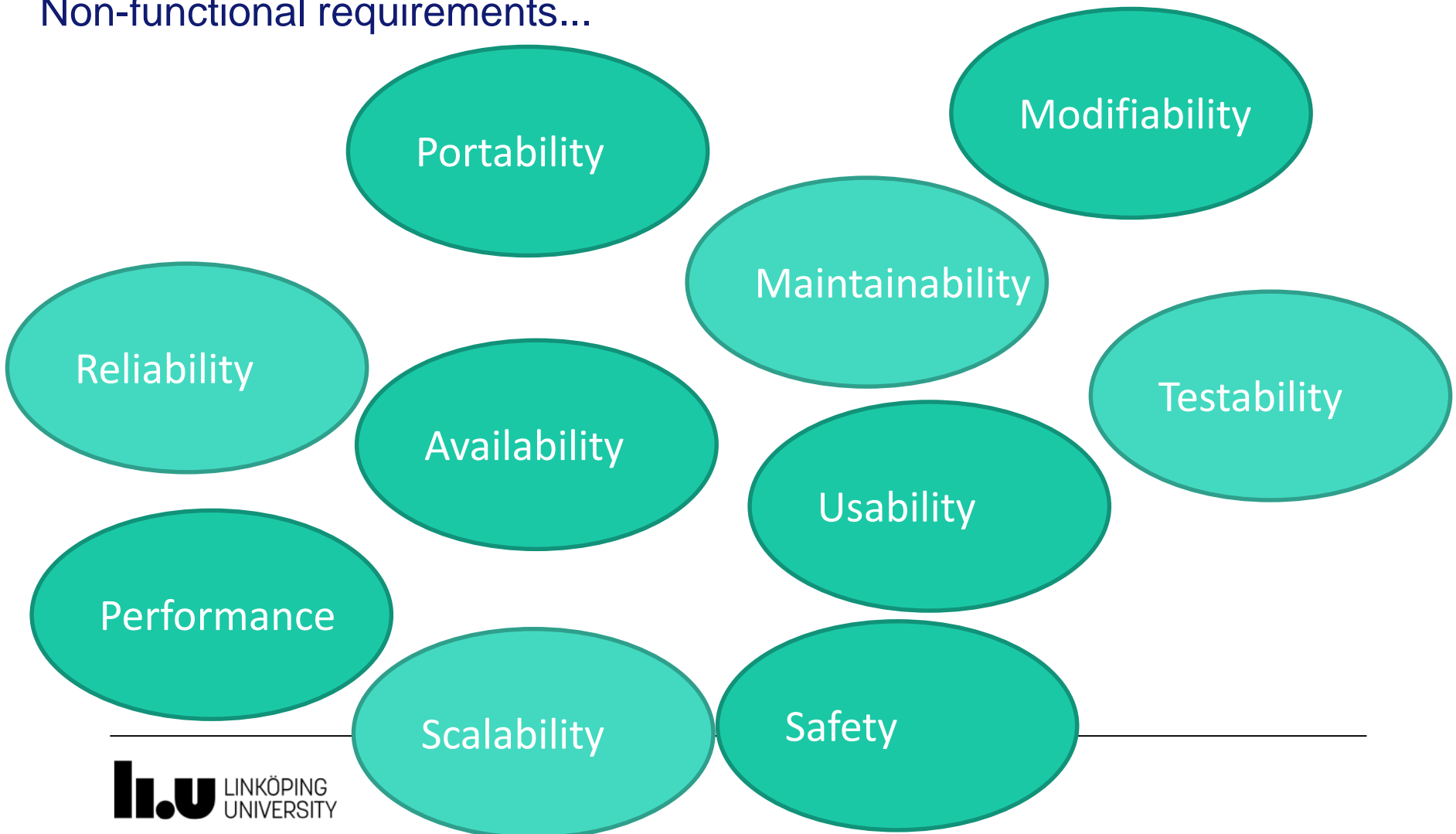
What do we want?

High cohesion. Why?

- More understandable
- Easier to maintain

Several factors - sometimes overlap

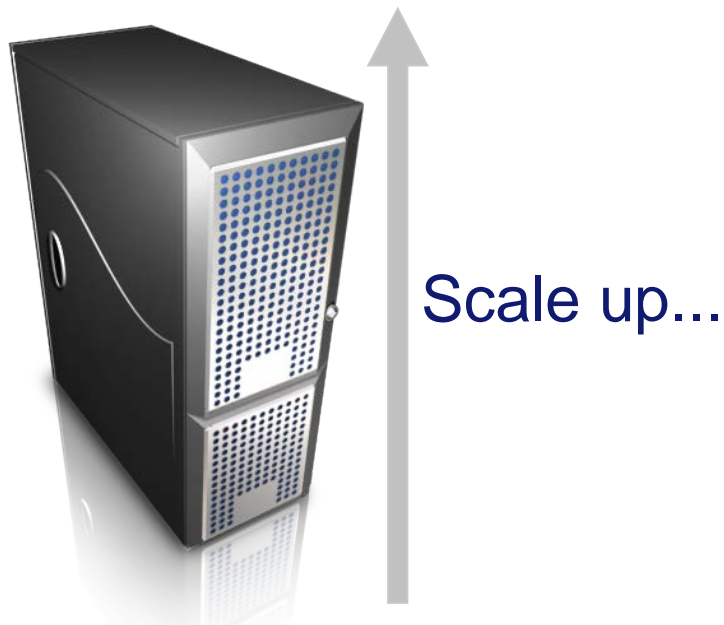
Non-functional requirements...



Performance - timing

Timing

- Throughput
- Response time (interactive system)



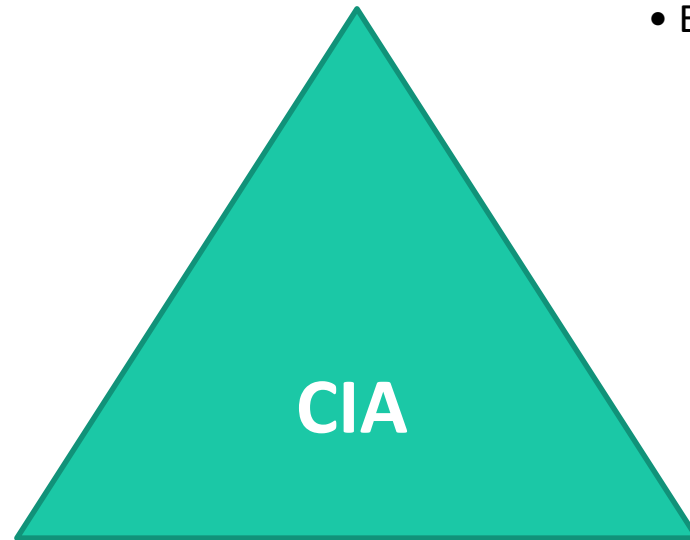
Can our architecture be parallelized?



Security

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

Safety - absence of critical faults

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

- Formal validation?
- Testing?

The whole system

Critical

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



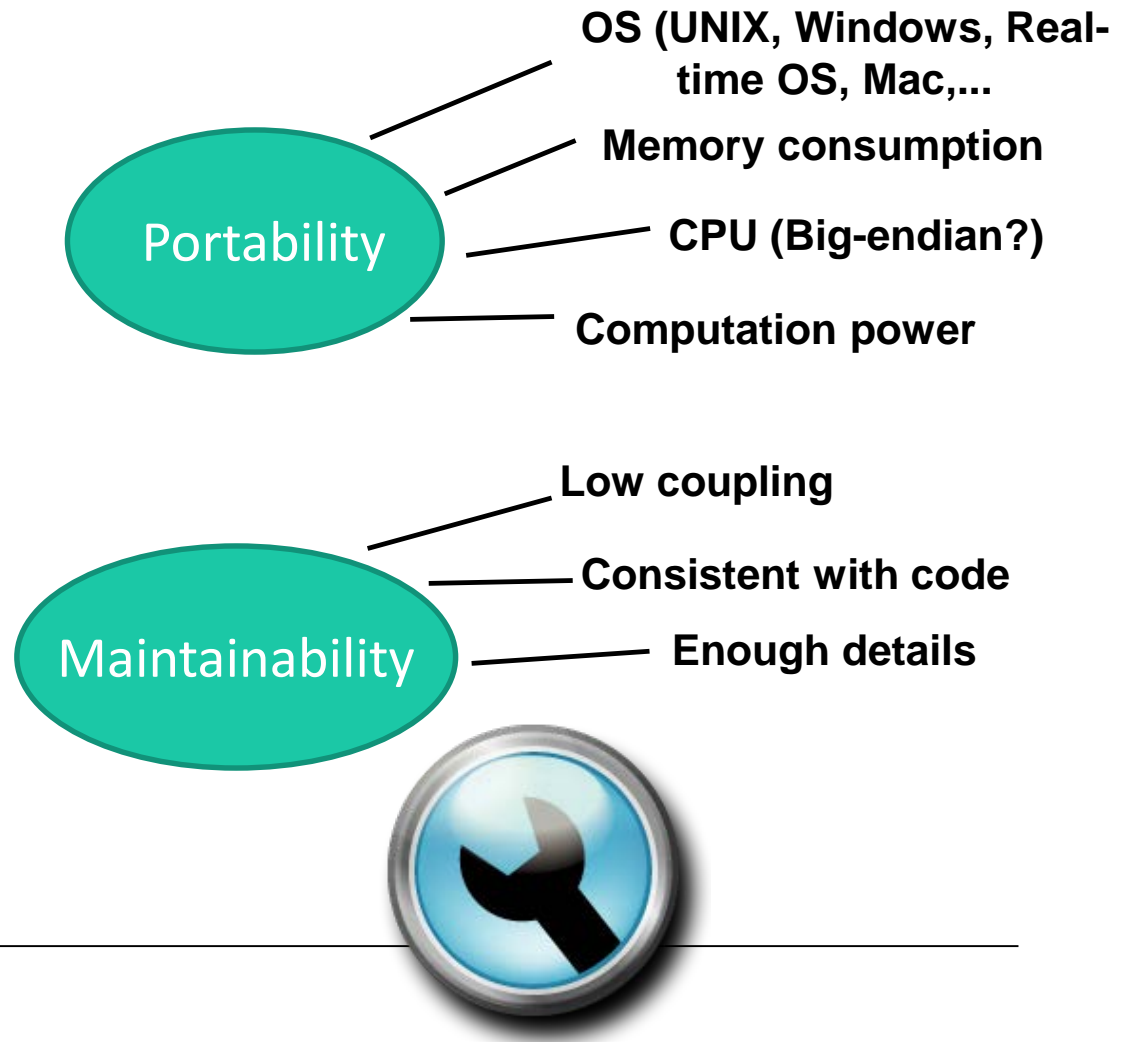
Modifiability - cost of change

What can change?


- Platform?
- Function?
- Protocols?
- Environment?

When can change?

- Source code?
- Compiler option?
- Library?
- Setup config?
- At runtime?

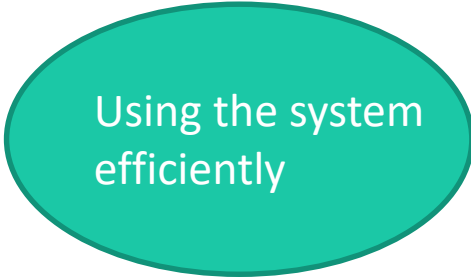


Usability - How easy is it and what support exists to perform a task



Easy to learn
system
features

E.g. a word-processor
or app

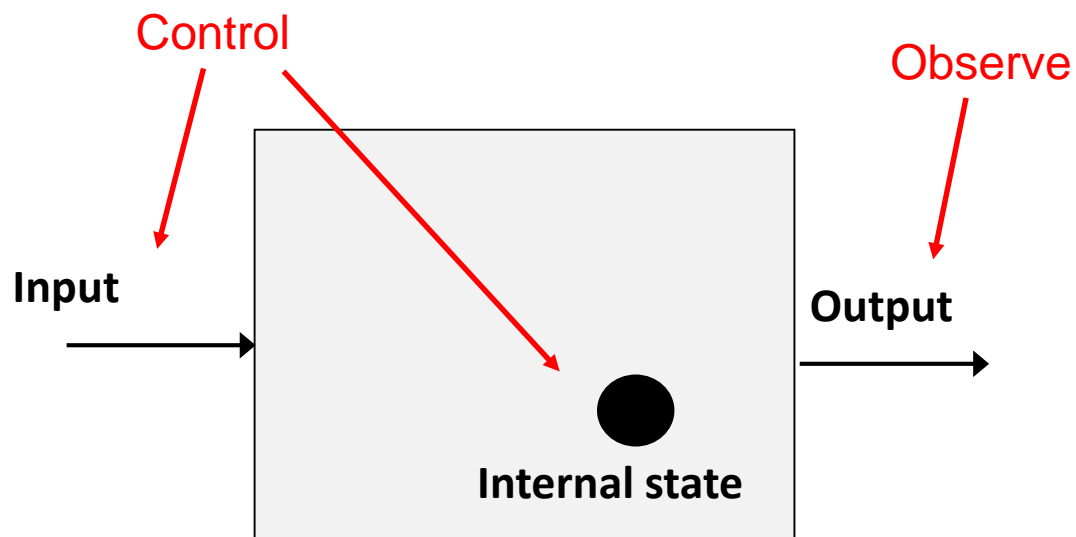


Using the system
efficiently

E.g. Latex, or UNIX
shells and pipes

Testability

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



What about cohesion and coupling?

Some Business Qualities



Time-to-market

Reuse component
and use commercial-
off-the-shelf (COTS)
products



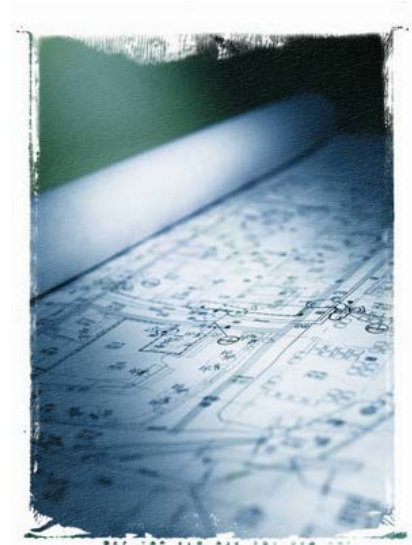
Cost-and-benefits

Use technology that
the organization knows

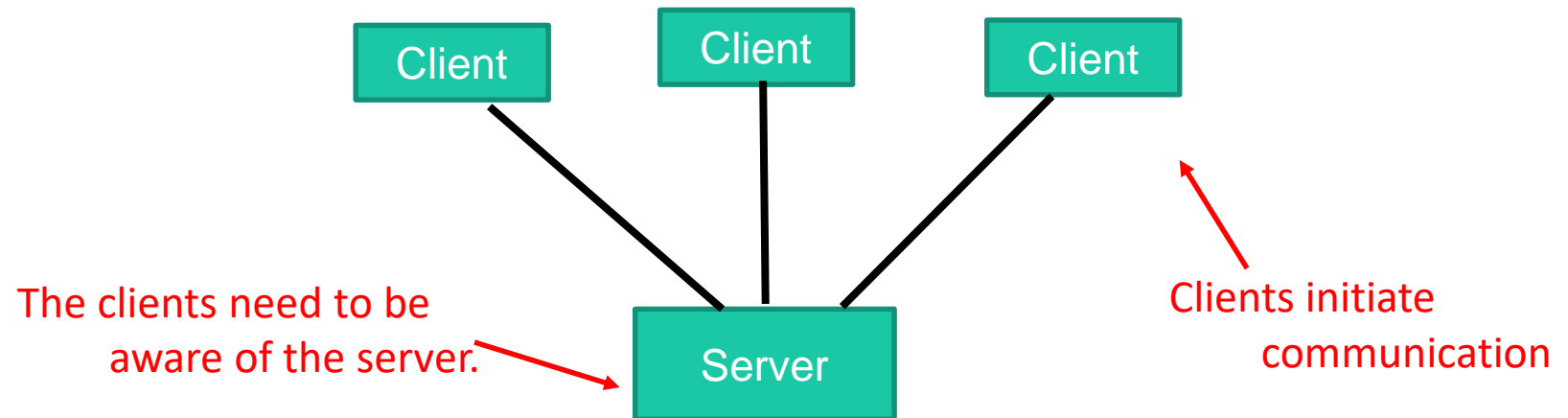
Architecture Styles / Patterns

Example of styles and patterns

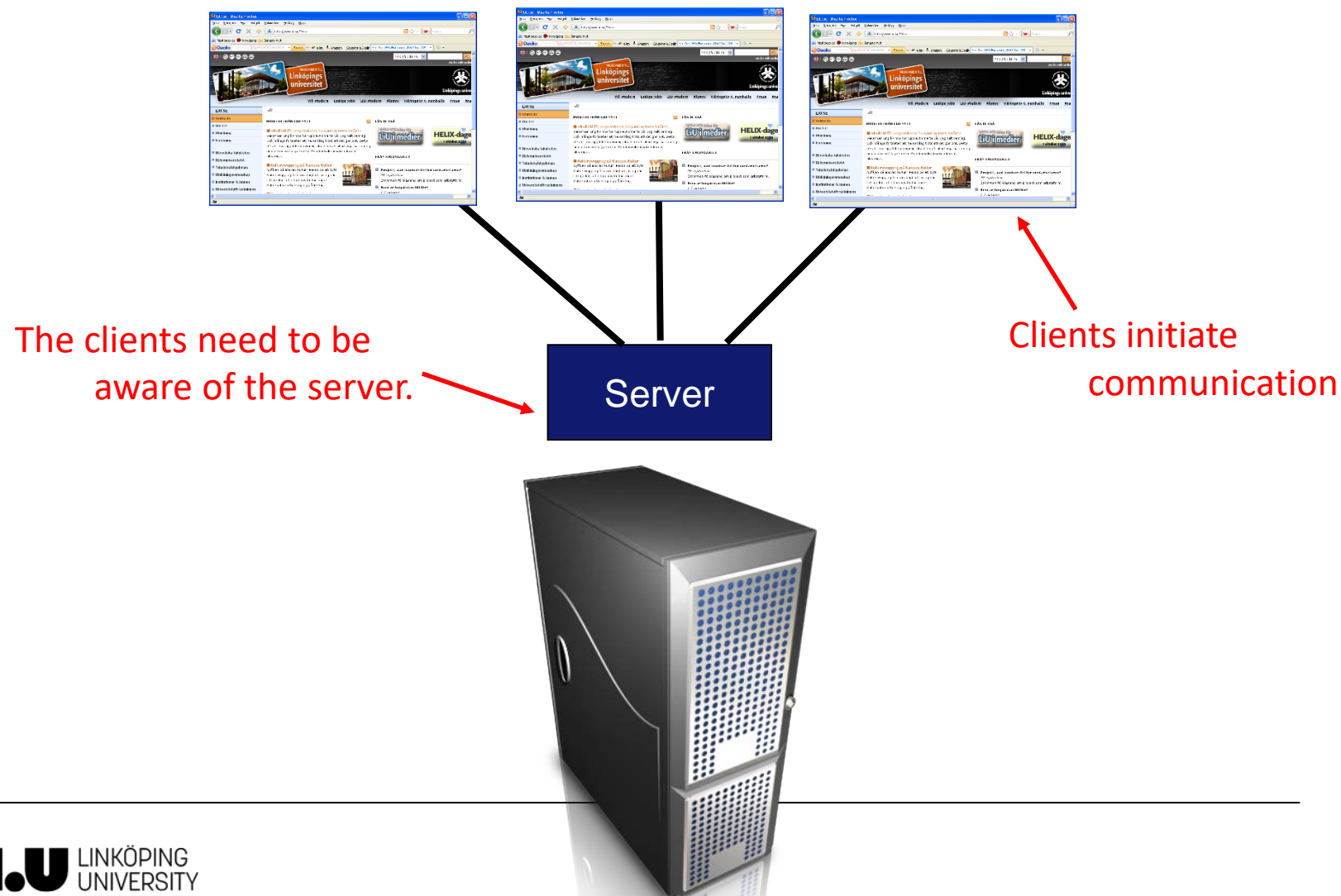
- **Client-Server**
 - **Layering**
 - **Pipes-and-filters**
 - **Service-oriented**
 - **Model-View-Control (MVC)**
 - **Repository**
 - **Peer-to-Peer**
- Discussed today



1. Client-Server

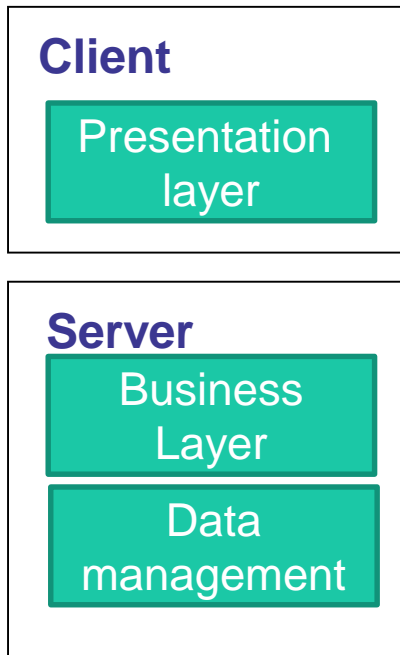


1. Client-Server



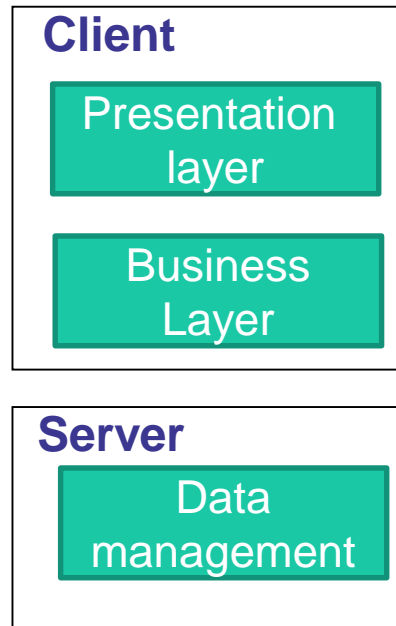
1. Client-Server

Two-Tier, Thin-client



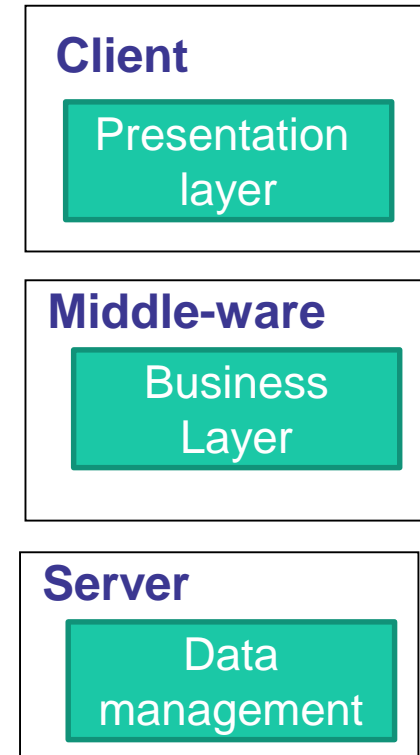
- Heavy load on server
- Significant network traffic

Two-Tier, Fat-client



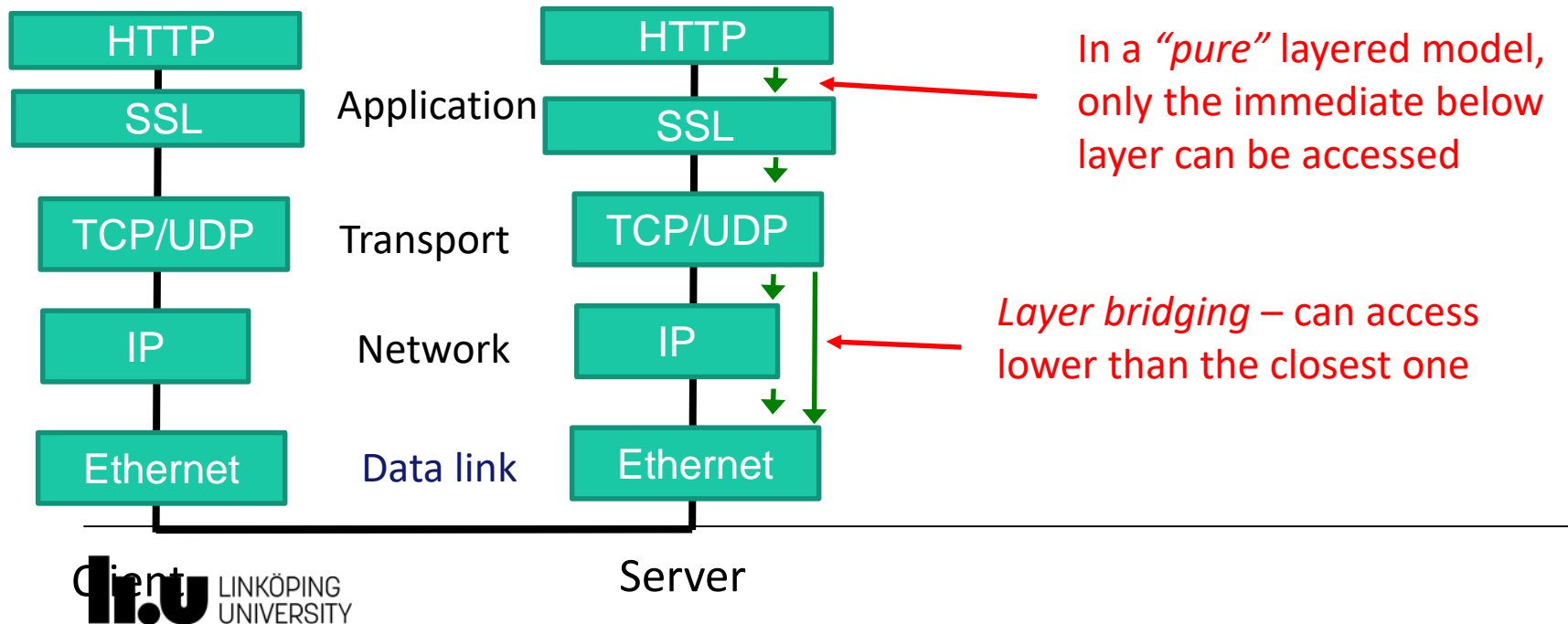
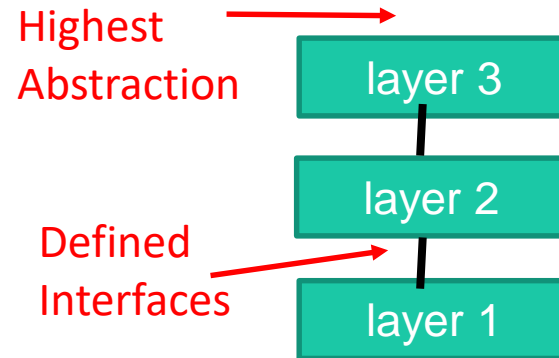
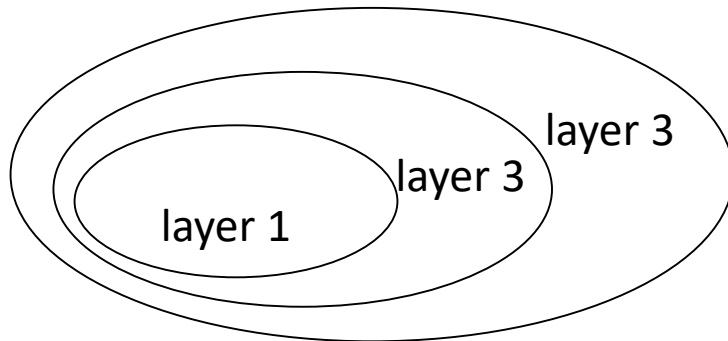
- + Distribute workload on clients
- System management problem, update software on clients

Three-Tier

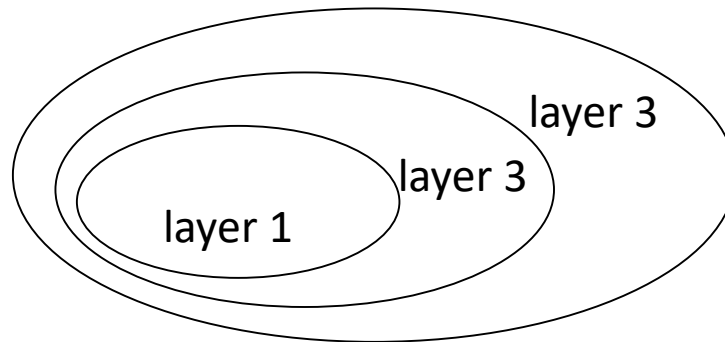


- + Map each layer on separate hardware
- + Possibility for load-balancing

2. Layers



2. Layers



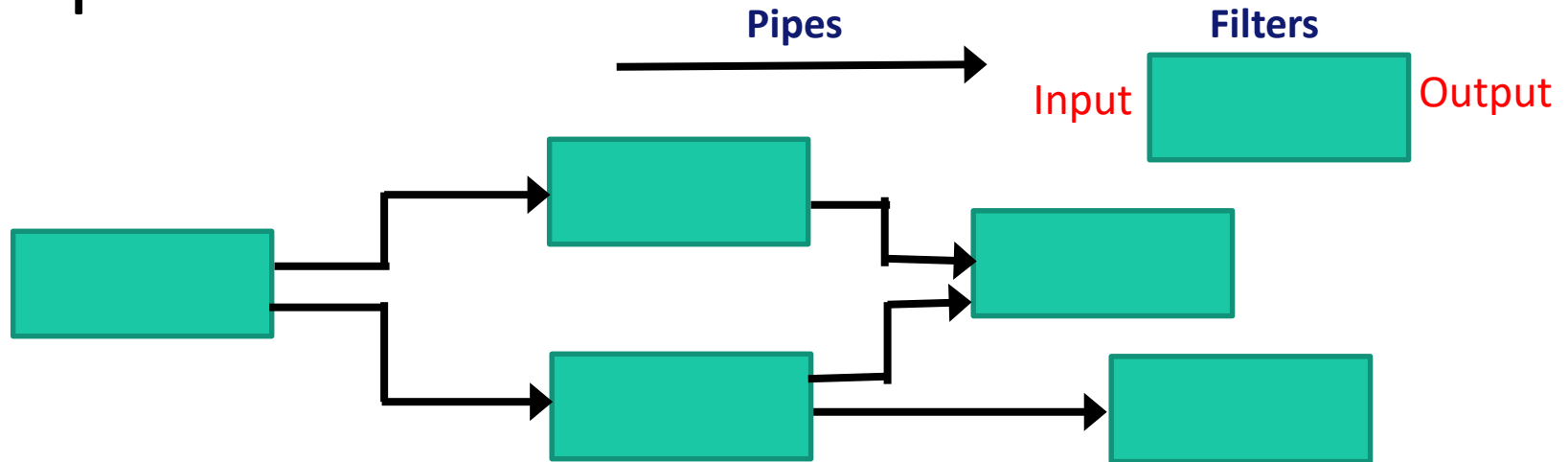
Pros

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

Cons

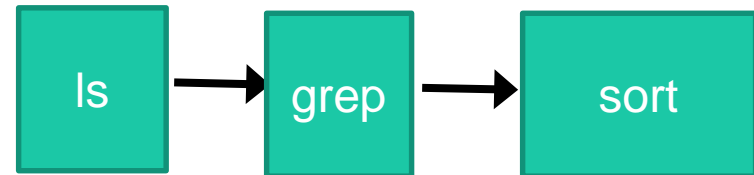
- Could give performance penalties
- Layer bridging looses modularity

3. Pipes and Filters

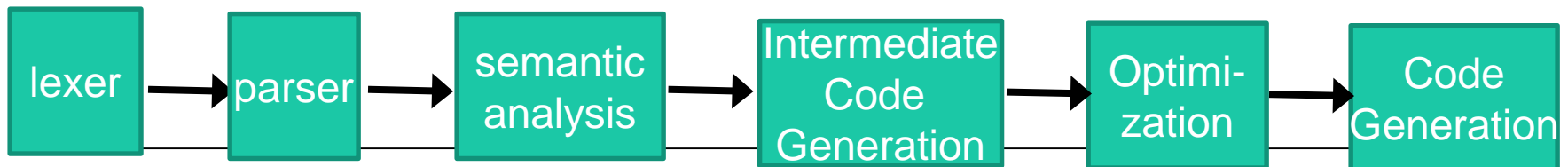


Example: UNIX Shell

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



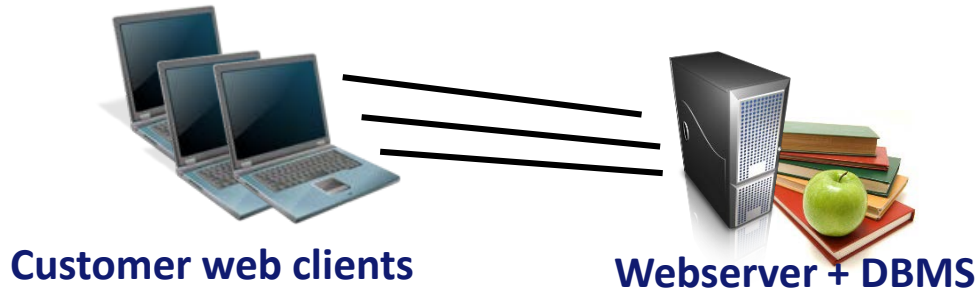
Example: A Compiler



Case: SOA and Amazon

Before 2001...

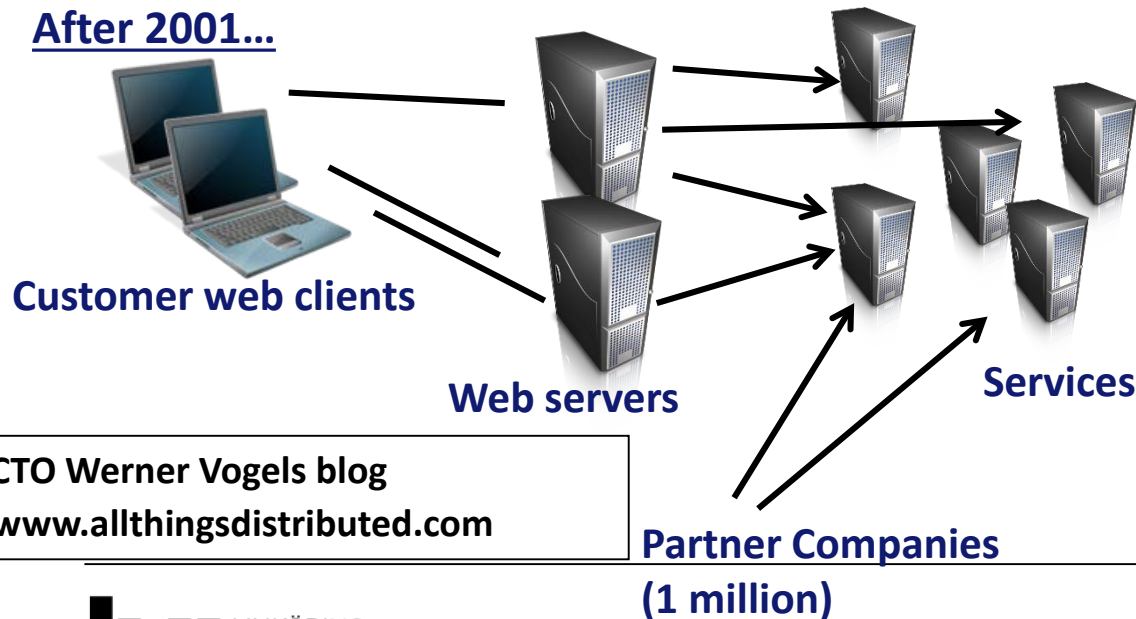
Two-tier architecture



Problems

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

After 2001...



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

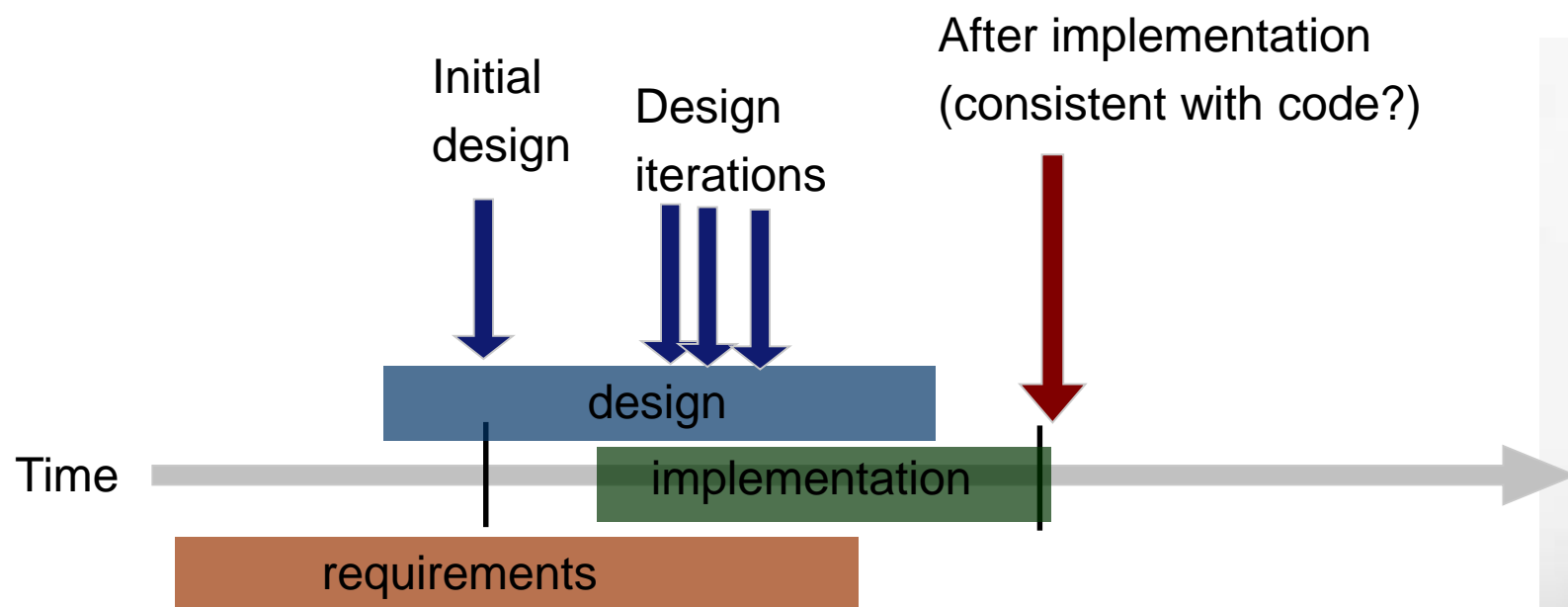
Coming back to documents...

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?



Software Architecture/Kristian Sandahl

www.liu.se