Tampereen yliopisto

# Large Scale Software Design

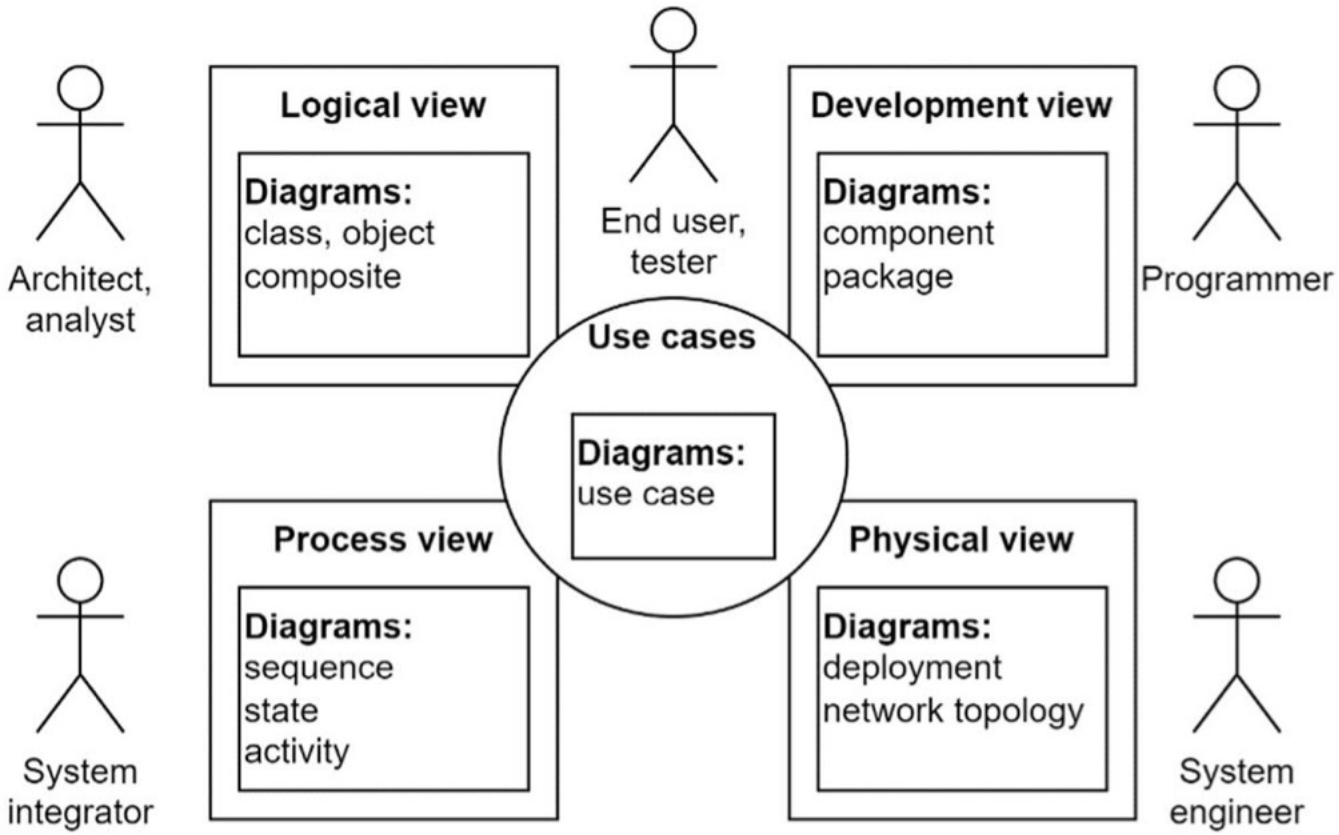
# Group assignment, facilitation 2

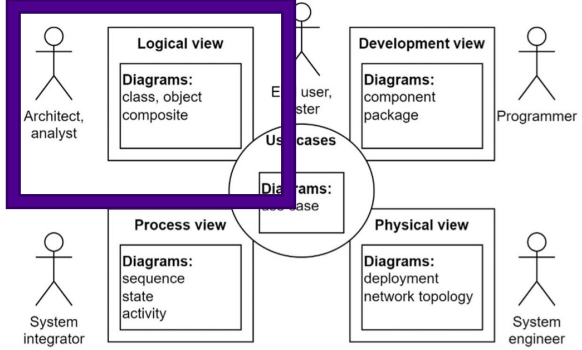
# Group assignment

- T-Low food delivery service
  - https://plus.tuni.fi/comp.se.210/spring2024/assignment

1. Intermediate
   - ~~Tuesday, 11th April 2024~~
   - Only requirements and use cases (one use case diagram is enough)
     - The scenarios are the +1 part of the 4+1 model
2. Final
   - Tuesday, 11th May 2024
   - Includes everything …also intermediate submission corrected after feedback!
   - Max. 4000 words for document
   - Two parts to return:
     - Assignment as PDF with diagrams embedded
     - 4 UML charts drawn also as separate submission (Papyrus files)

# 4+1

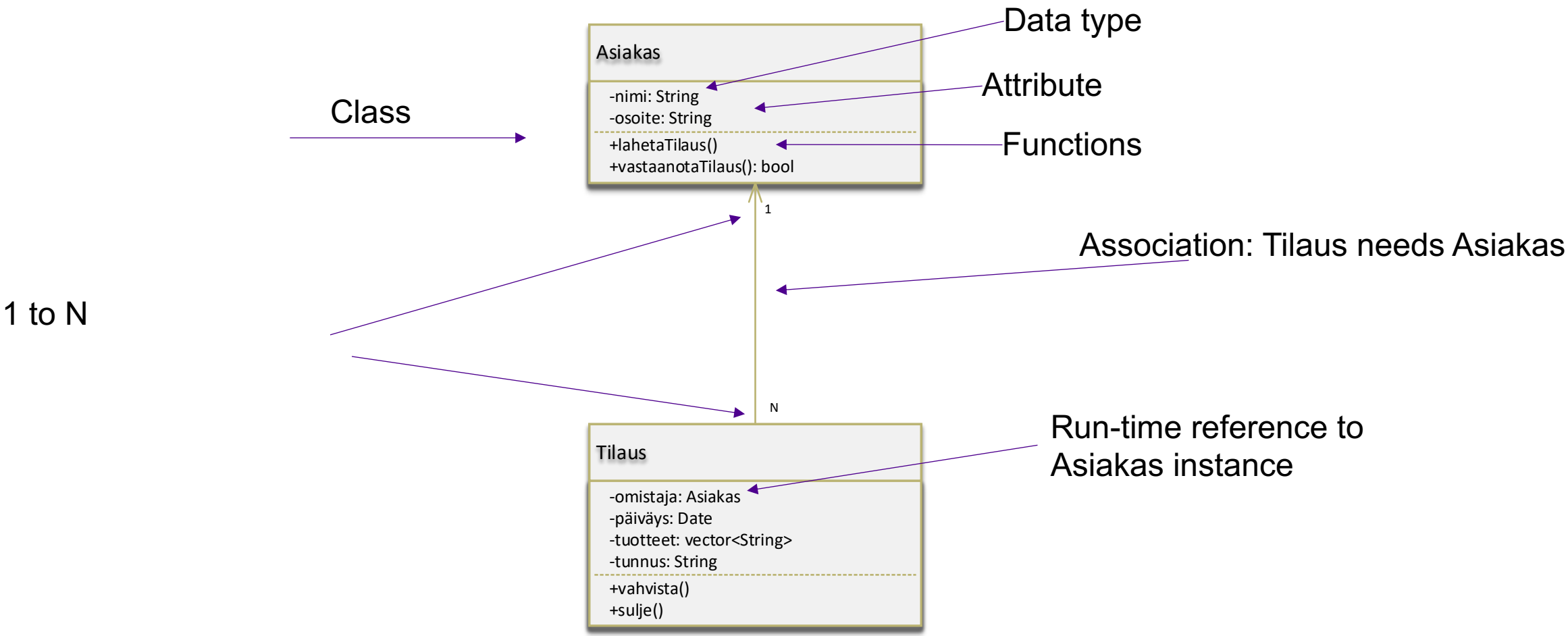

Logical view
Diagrams:
class, object
composite
Development view
Diagrams:
component
package
Use cases
Diagrams:
use case
Process view
Diagrams:
sequence
state
activity
Physical view
Diagrams:
deployment
network topology
Architect,
analyst
End user,
tester
Programmer
System
integrator
System
engineer




# UML – Class diagram

- Static structure, can be used as a blueprint to the structure of the code
- Contains the attributes and operations
- Shows classes, interfaces, associations, collaborations etc.
- Illustrates the responsibilities
- Can be a starting point for component diagrams and deployment diagrams

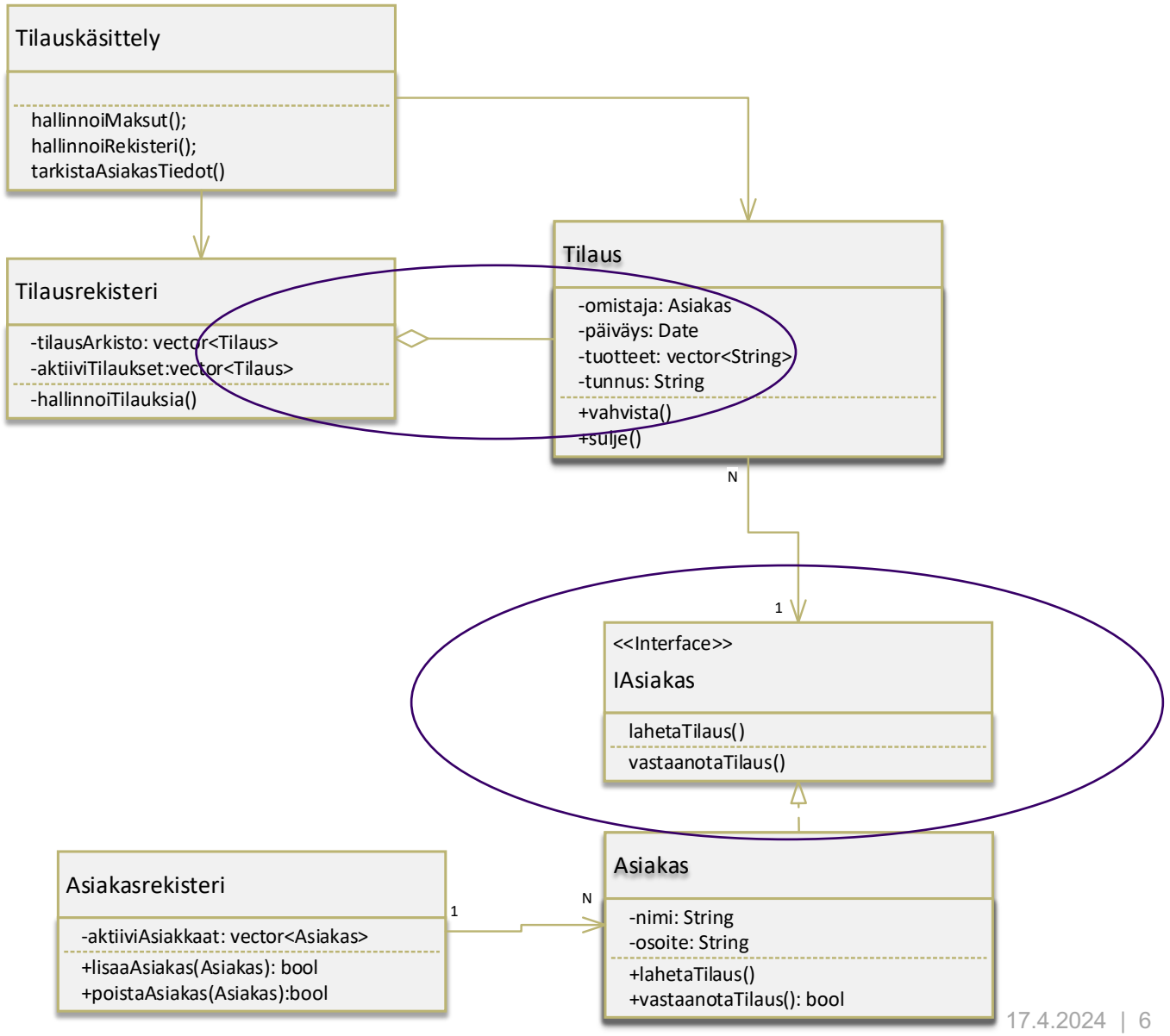
# UML – Class diagram

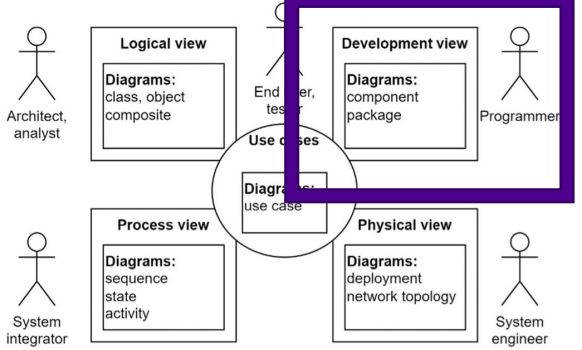


# Class diagram cont.

- Aggregation, composition

- Interface

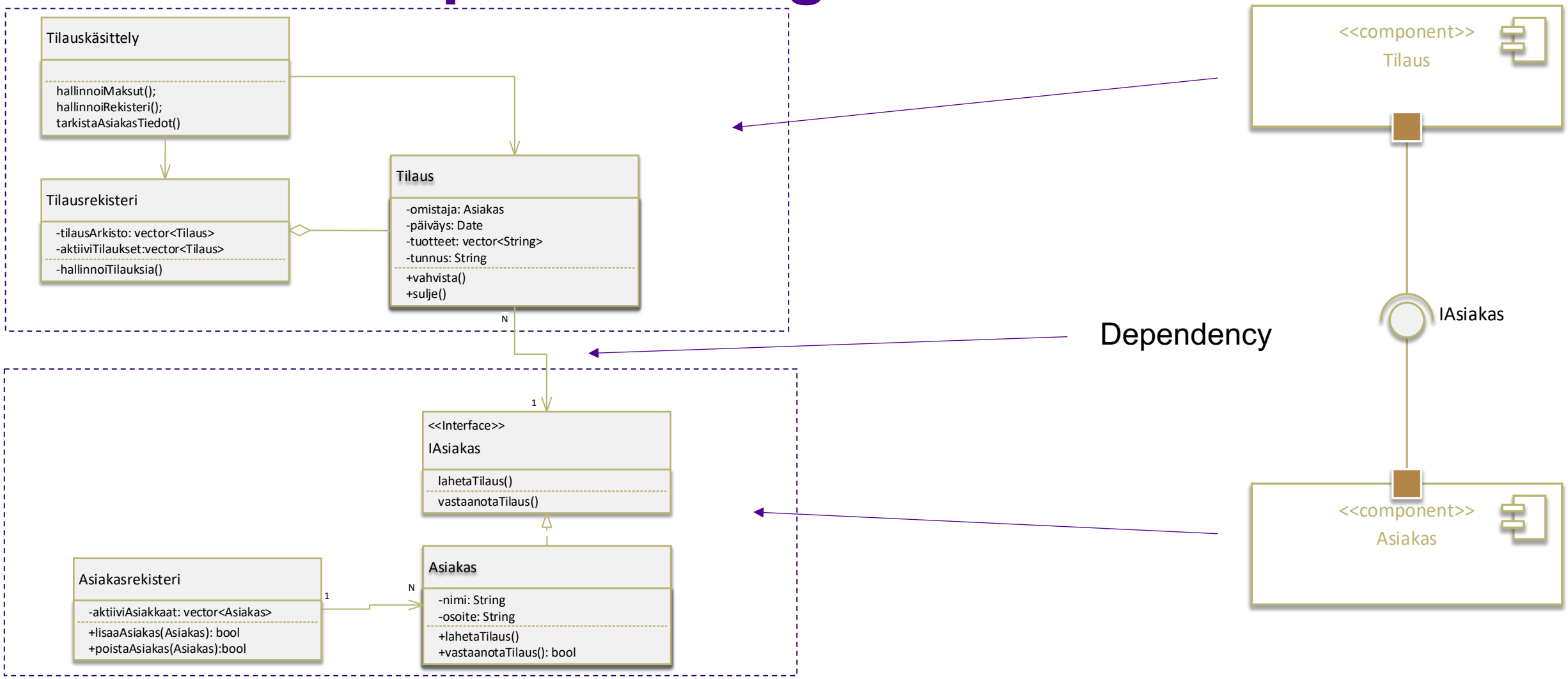


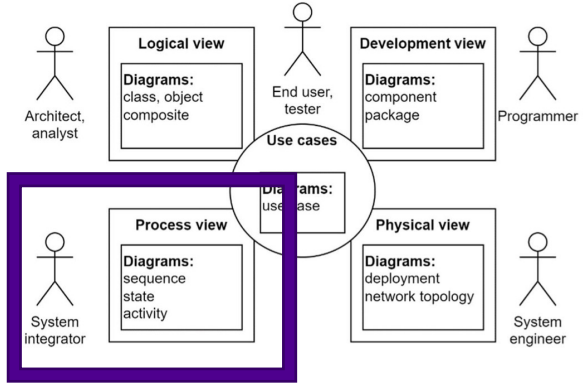



# UML – Component diagram

- Static view (4+1 → logical view – physical view?)
- Higher level representation than a class diagram
- A component often is a class module that could be extracted and added to another system, and used through an interface

# UML – Component diagram

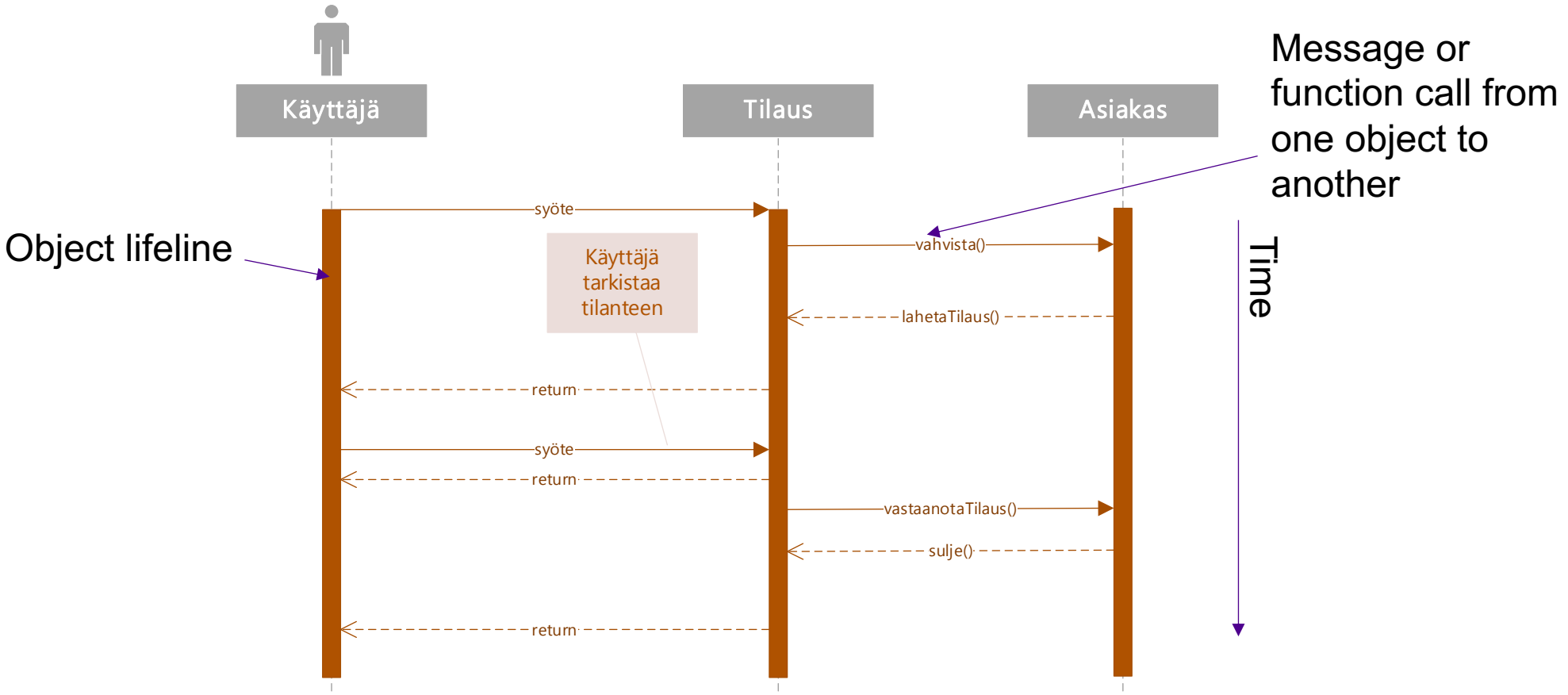


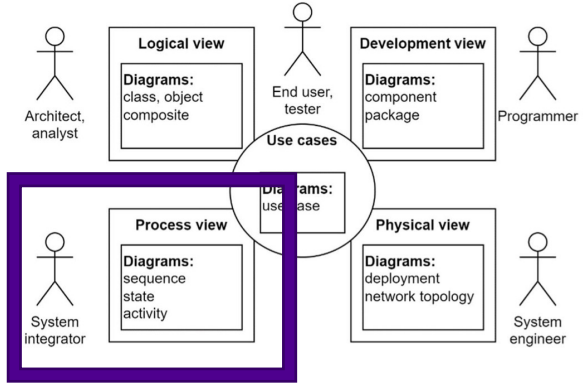



# UML – Sequence diagram

- Dynamic behavior (4+1 → Process view)
- Illustrates the interaction between objects or information flow between objects
- Illustrates the order of operations
- Dimensions are objects and time

# UML – Sequence diagram





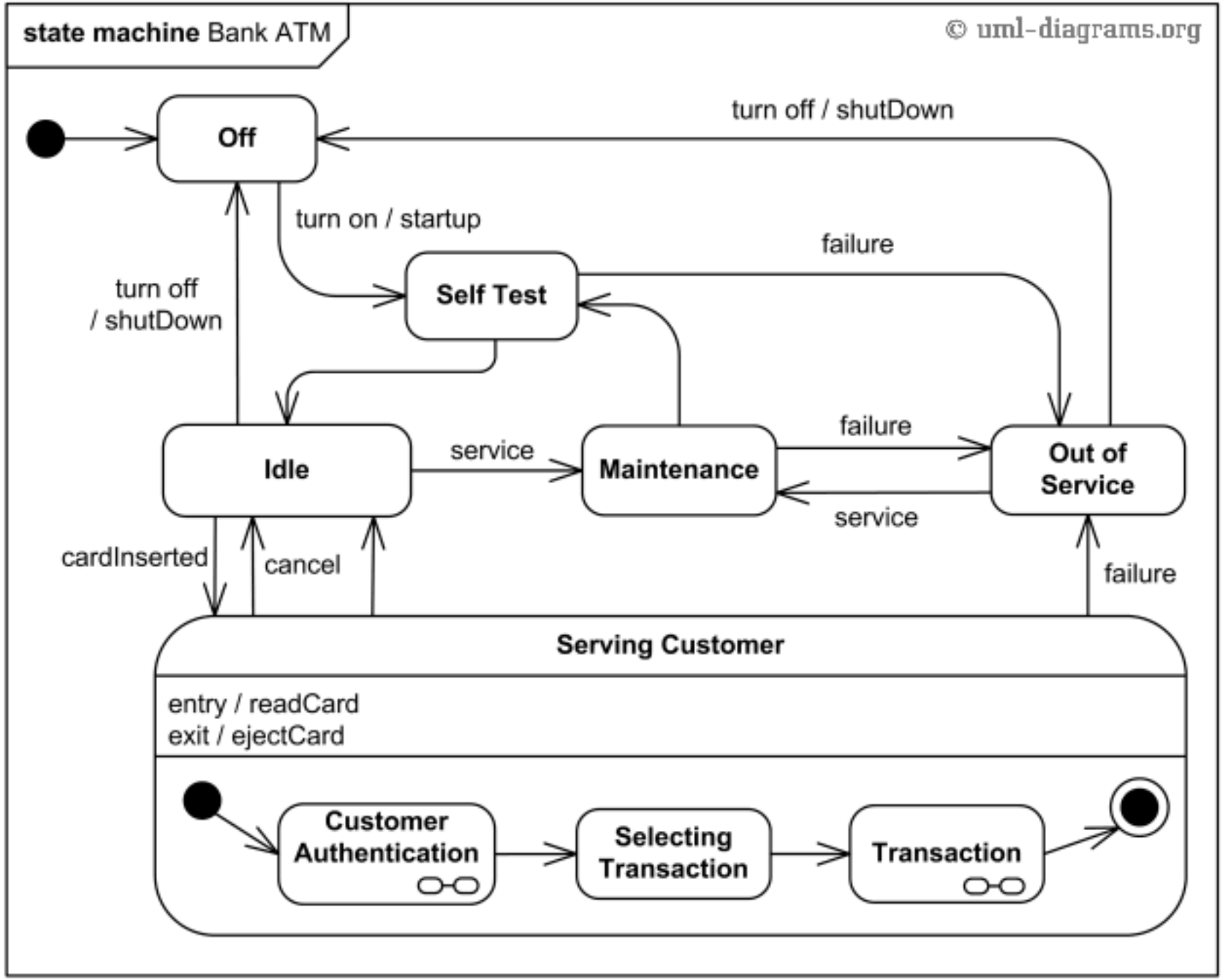

# UML – State diagram

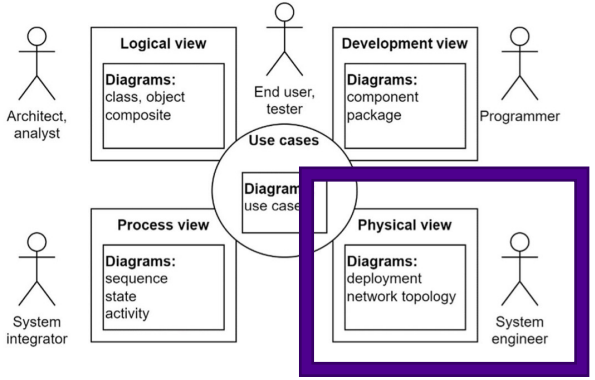
- Dynamic behavior (4+1 → Process view)
- Object state changes represented as a state machine
  - Events trigger state transitions and produce actions
- The behavior of individual objects can differ in different situations
- The object behavior depends on its state

# UML – State diagram

State diagram vs sequence diagram?

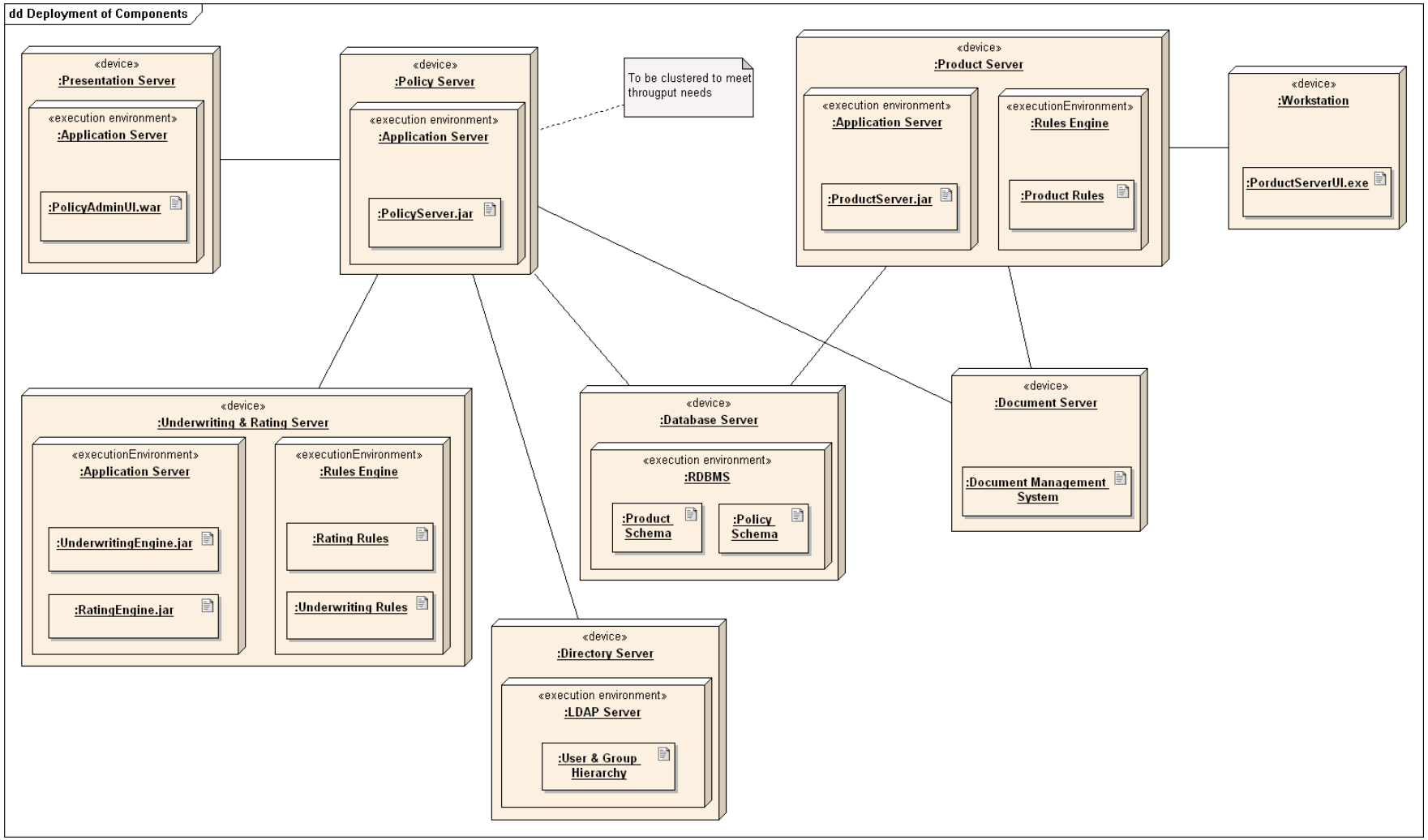
State diagram vs class diagram?



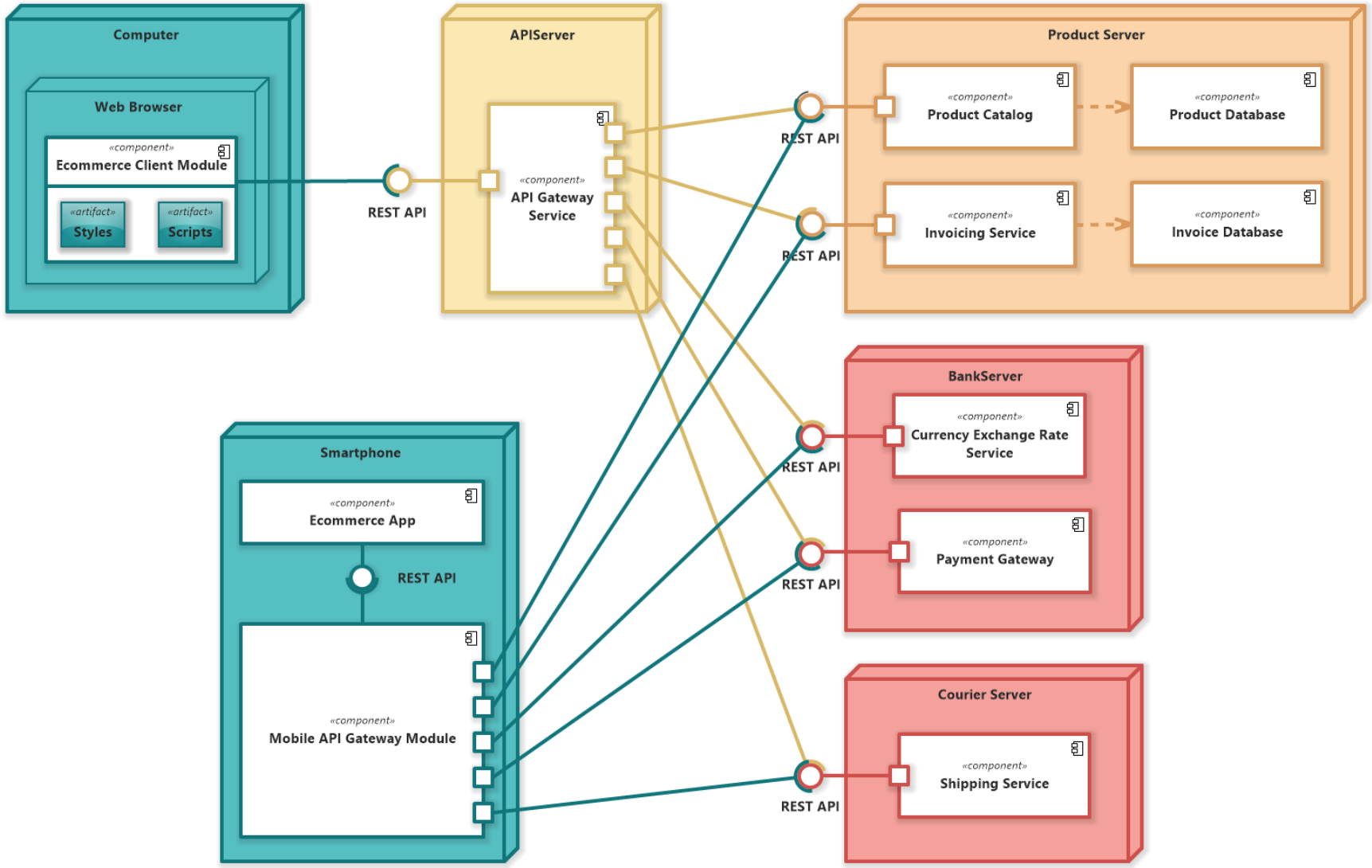



# UML – Deployment diagram

- Physical view
- Models the physical deployment of artifacts



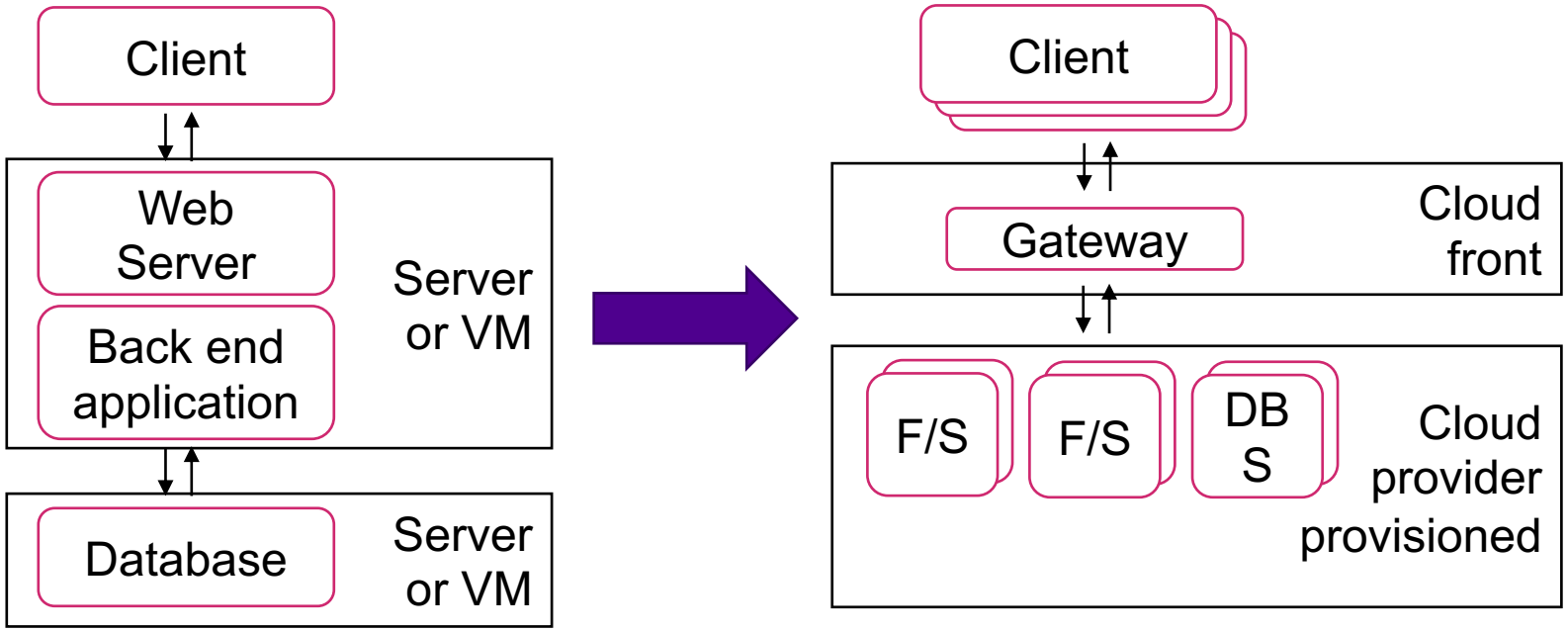
# UML – Deployment diagram



# “Things to consider when developing cloud-native architectures”

- Microservices, serverless
  - Technologies of cloud providers
- Information security
- Interoperability

# Back-end evolution from monolith applications to multiple components

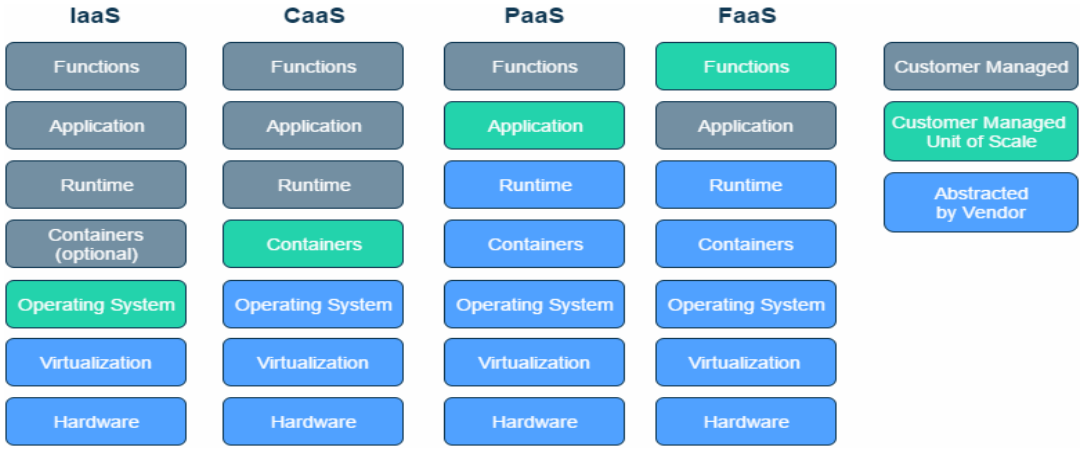


# The evolution - the big picture for software, IaaS vs CaaS vs PaaS vs Faas

Monoliths, modular software, …

Microservices, containers

Functions, containers and less



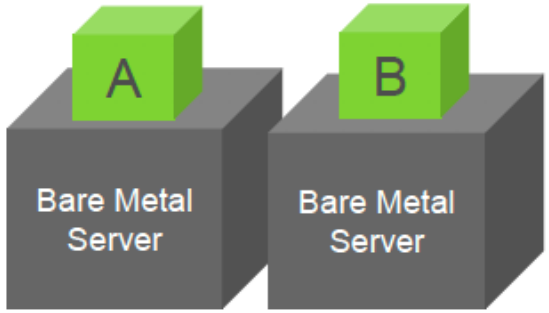

Less infrastructure concerns

Less control and more dependencies

# Run only on demand

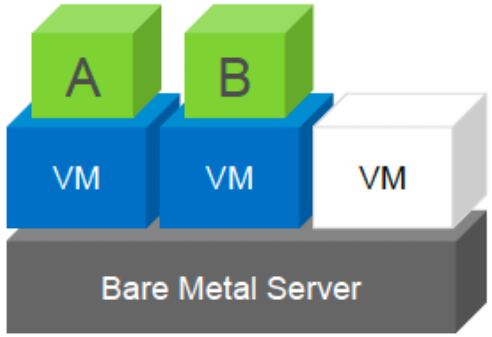
1 → 2 → 3 → 4

## 1 Dedicated Server



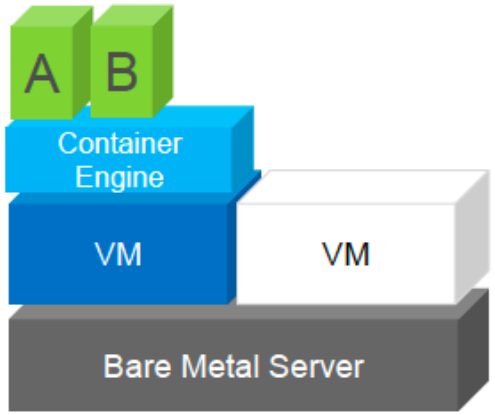

In case of dedicated servers applications (A, B) are deployed on physical servers. In consequence, the servers are often over dimensioned and have inefficient utilization rates.

## 2 Virtualization



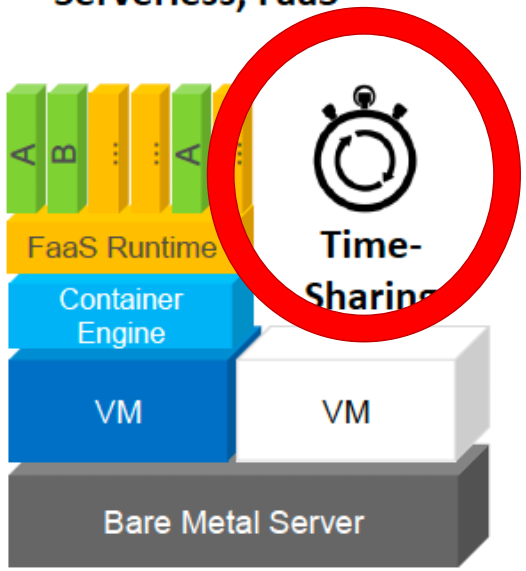

Machine virtualization is mainly used to consolidate and isolate applications on virtual machine instead of dedicated servers. This increases the application density on bare metal servers but the virtual machine images (deployment unit) are very large.

## 3 Containerization Microservices




To pragmatically operate more than one application per virtual machine, containerization established as a trend. A container starts faster than a virtual machine and shares the operating system with other containers, thus reducing deployment unit sizes and increasing application density per virtual machine.

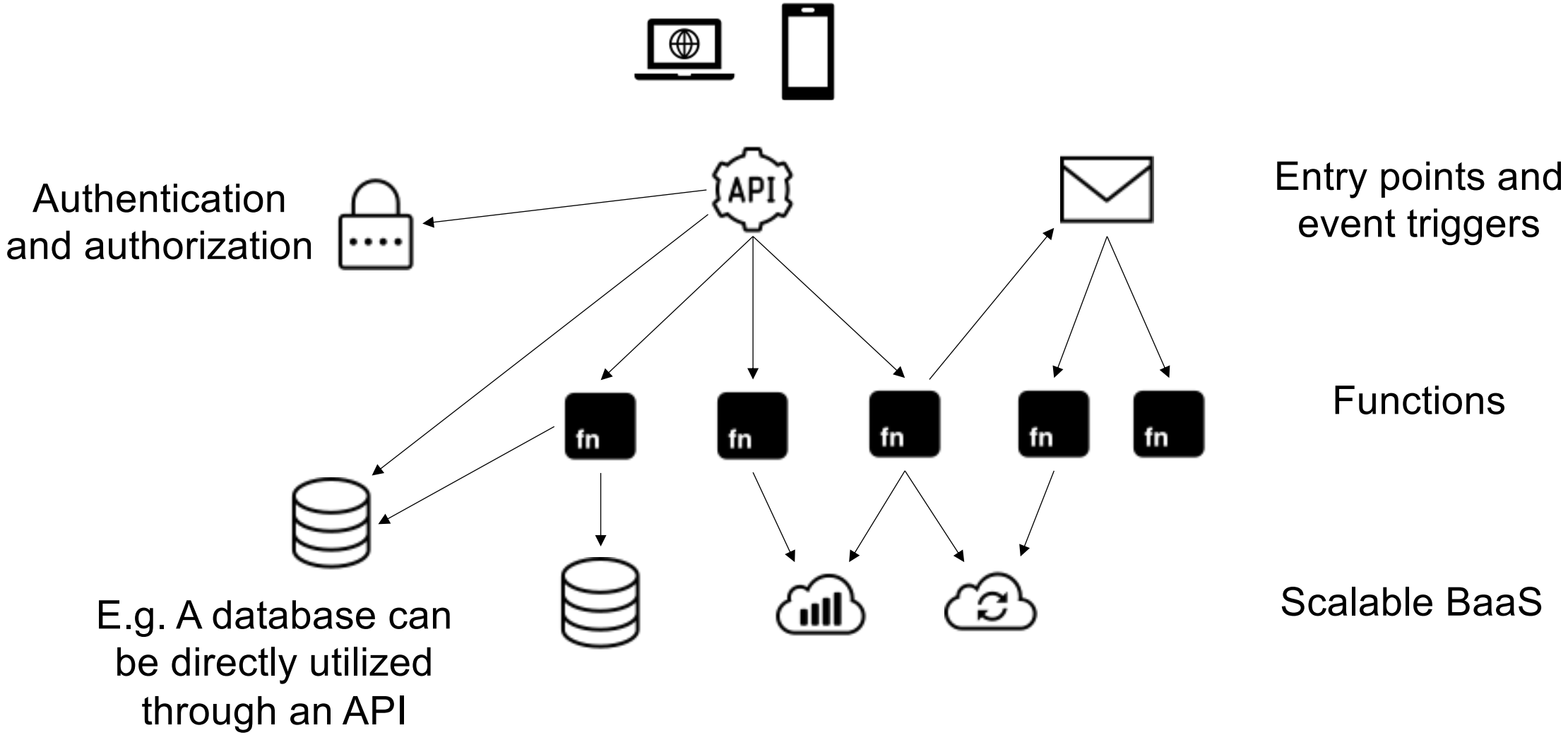
## 4 Serverless, FaaS




But a container still requests a share of CPU, memory, and storage – even if the provided service is hardly requested. It is more resource efficient, if services would consume resources only if there are incoming requests. FaaS runtime environments enable that services can timeshare a host. However, this involves to follow a serverless architecture style.

Nane Kratzke, A Brief History of Cloud Application Architectures From Deployment Monoliths via Microservices to Serverless Architectures and Possible Roads Ahead - A Review from the Frontline. Appl. Sci. 2018, 8, 1368

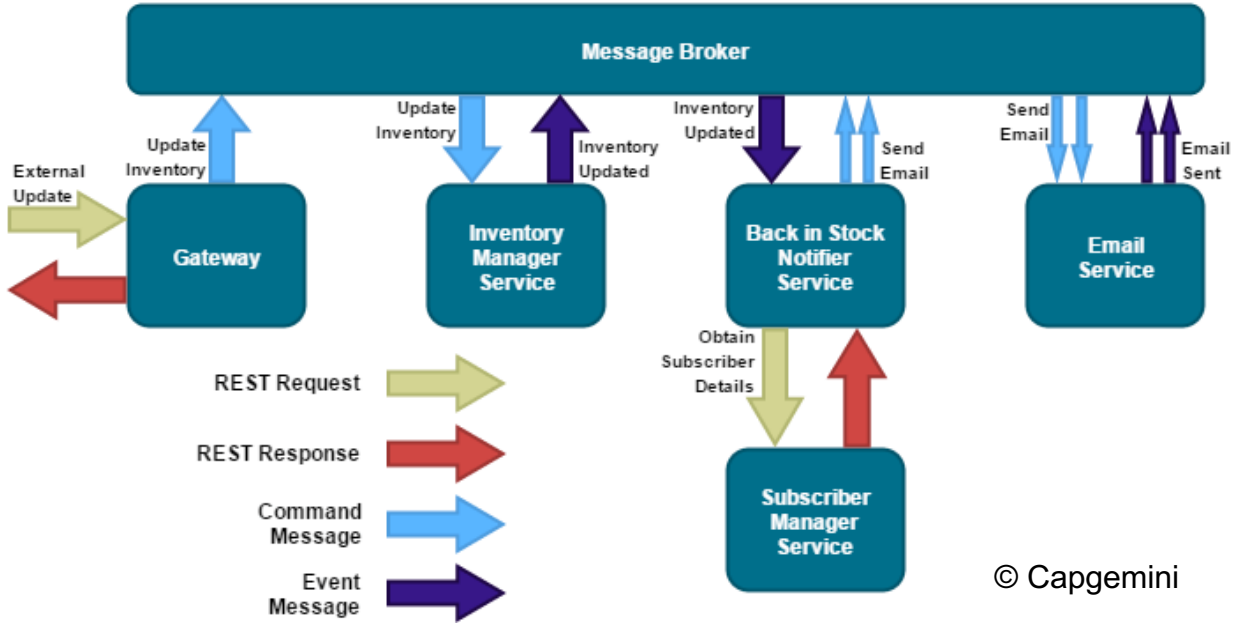
# Serverless architecture



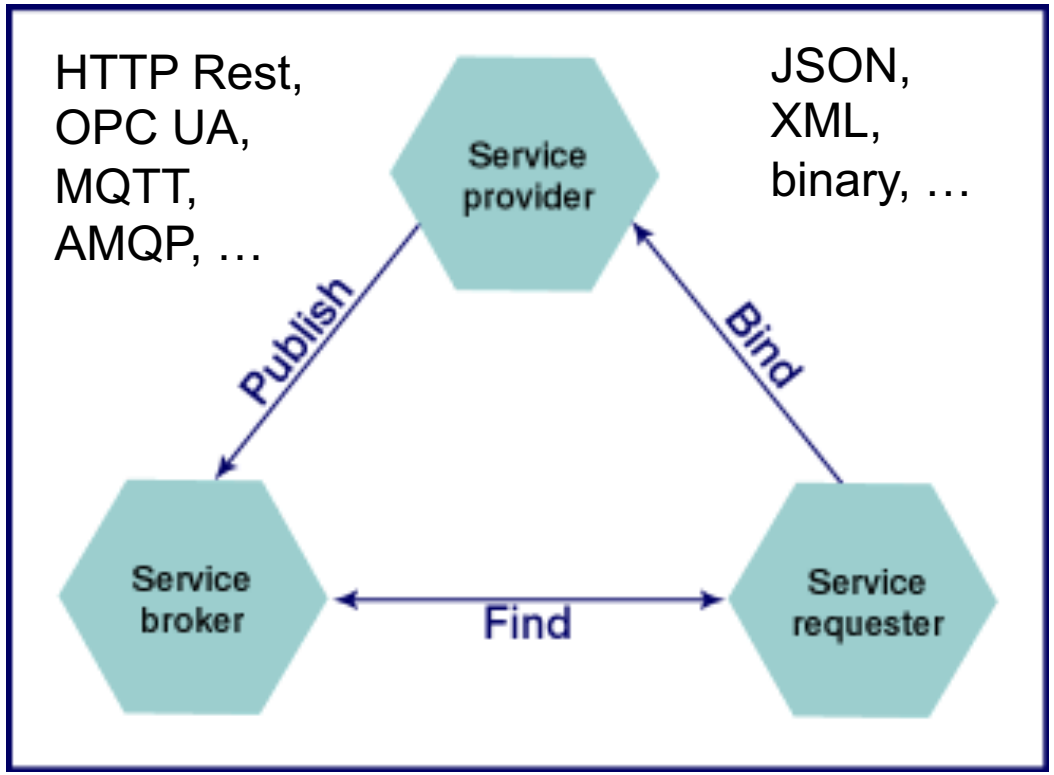
# Communication between components

# Message brokers (message queue, message bus)

- Can act as an intermediary between two communicating parties
- Asynchronous by nature
- Patterns:
  - Streams/queues
  - Publish-subscribe
- May support HTTP REST, SOAP or any other means for clients/application components to connect



# Service composition compatibility




More protocol agnostic

# Asynchronous communication

- Asynchronous communication is often necessary on the web
  - HTTP requests have a time limit
  - Activities may actually take long (e.g. hours)
  - We do not have control of the other end or its load

- Three common ways to implement asynchronous communication using HTTP
  - Polling - sending requests at a consistent rate (e.g. every 5s) and the server returns updates accordingly (also no news updates)
  - Long-polling - the client sends a request which is immediately responded, followed by another request for which the server returns a response when something new is available
  - Push - after a request is being sent the response by the server is kept open, i.e. sending sub messages to keep the connection open indefinitely (also pings might be necessary to prevent the browser from closing the connection due to time-outs)
- Remember: application layer vs protocol layer differences!

# Other means of implementing asynchronous communication

- Duplex communication (as HTTP is not bidirectional)
  - APIs in both directions
    - Challenging for web client applications (in the browser) as they cannot be invoked using HTTP
    - But between application components (server side) and different systems this is less of an issue
- Not using HTTP
  - E.g. TCP instead
  - Using some application layer on top of HTTP (e.g. using a library such as SignalR)
- Mediator approach, i.e. having another component in between
  - E.g. job que, message bus, broker, …

# Information security

# What is information security?

- The classic model for information security (mozilla.org):
  - **Confidentiality** refers to protecting information from being accessed by unauthorized parties. In other words, only the people who are authorized to do so can gain access to sensitive data.
  - **Integrity** refers to ensuring the authenticity of information—that information is not altered, and that the source of the information is genuine
  - **Availability** means that information is accessible by authorized users
- Each have a different viewpoint to protection of information

# Web Applications Security considerations

- Protect data stored, e.g. encryption or restricted access to storage
- Protect communication
  - Between client and server, e.g. TLS
  - Between distributed components, e.g. TLS
- How to ensure who sends or requests data?
  - Identify each application with authentication, e.g. user name and password
  - Certificates or tokens etc. to sign that data comes from a certain party (and that is hasn't been altered on the way)
- Distributed software components and services
  - Own backend or distributed services around the Internet?
  - Can you rely on the "local" network, e.g. docker environment?
- Intermediary components such as a message bus - Can you trust that only authorized parties access that service? I.e. messages received are from who they are?
- Keep an inventory of your web applications, maintenance needs, dependencies, and monitor for vulnerabilities