# 3.7 Architecture Diagrams

Subject/Topic/Focus:

- ❏ Introduction to Architecture Design

Summary:

- ❏ Package Diagrams:
  - Classes, Interfaces & Packages
  - Nested Packages
  - Dependencies
- ❏ Deployment Diagrams:
  - Nodes & Components

Literature:

- ❏ [Fowler97]
- ❏ [Booch98]



# Architecture Design Models

An **architecture model** (structure model) is a model of a data processing system describing the static structure of the components of a system.

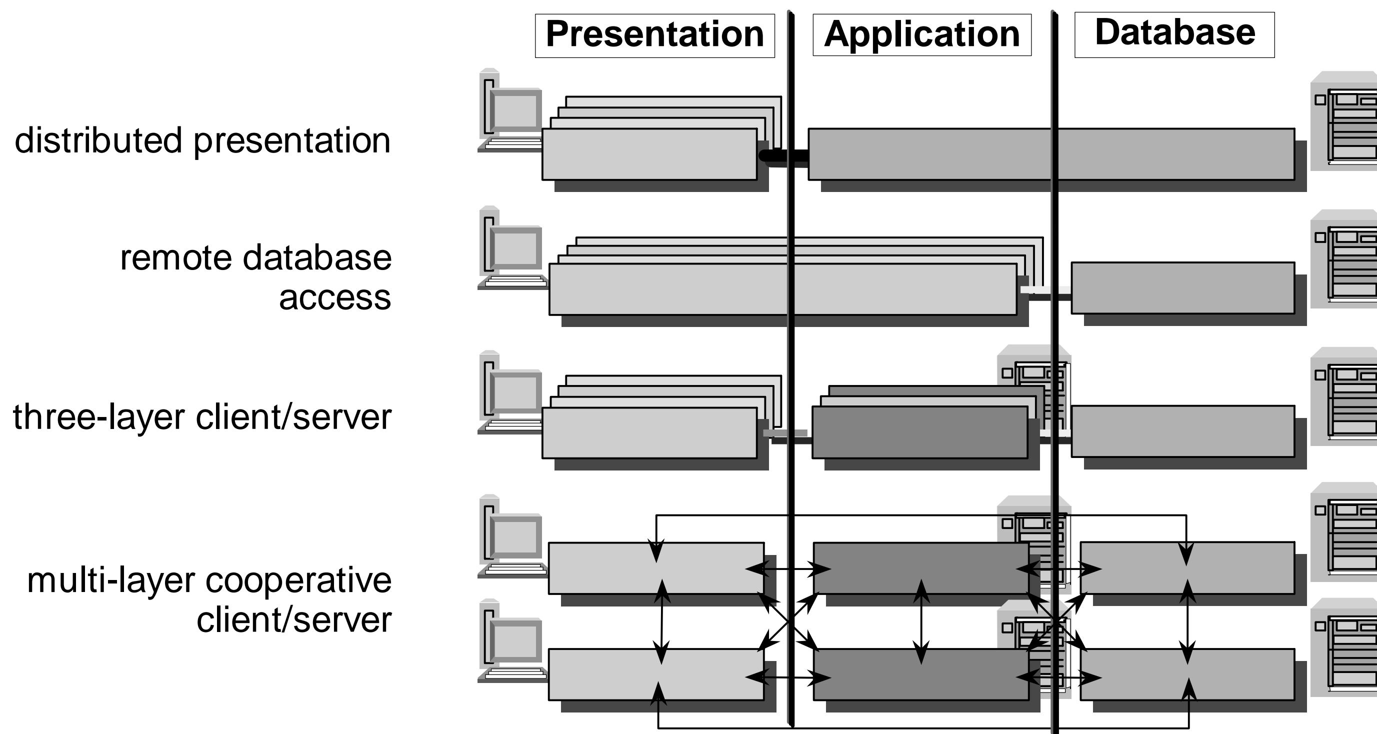
Examples:

- ❏ network topology (hardware)
- ❏ block schema (hardware)
- ❏ function tree (software)
- ❏ deployment diagram, package diagram (software)
- ❏ module diagram (hard/software)
- ❏ organization chart in a company model

© J.W. Schmidt, Florian Matthes 1999

# Example: SAP R/3

Flexible three-tier *client/server*-architecture






# Specification Models in UML

Specification models for architecture design in UML:

❑ **logical** structures

- packages, package diagrams
- subsystems
- interfaces

❑ **physical** structures

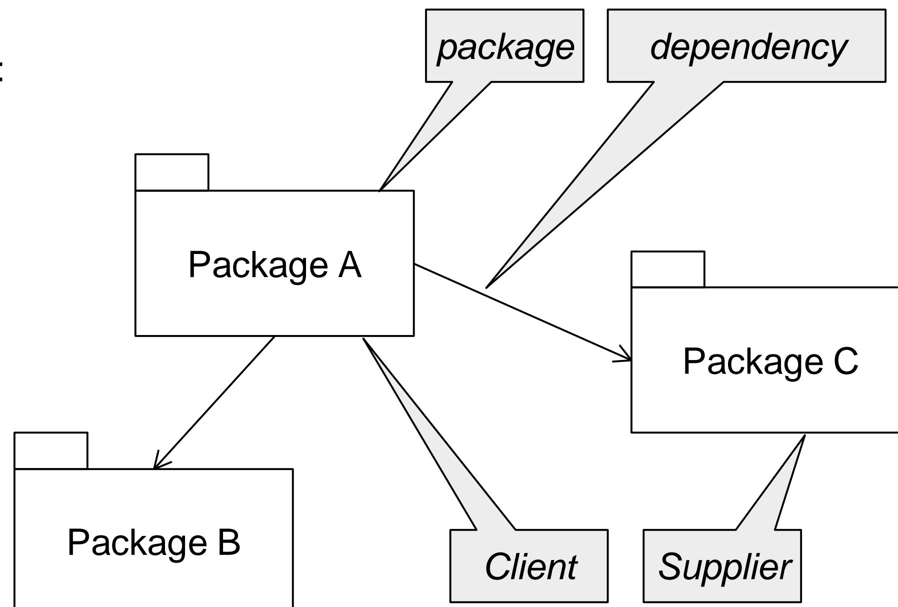
- components
- component diagrams
- nodes
- deployment diagrams

© J.W. Schmidt, Florian Matthes 1999

# Package Diagram

A **package diagram** shows the coherence (*dependencies*) between different *packages* of the system.

**Notation**:





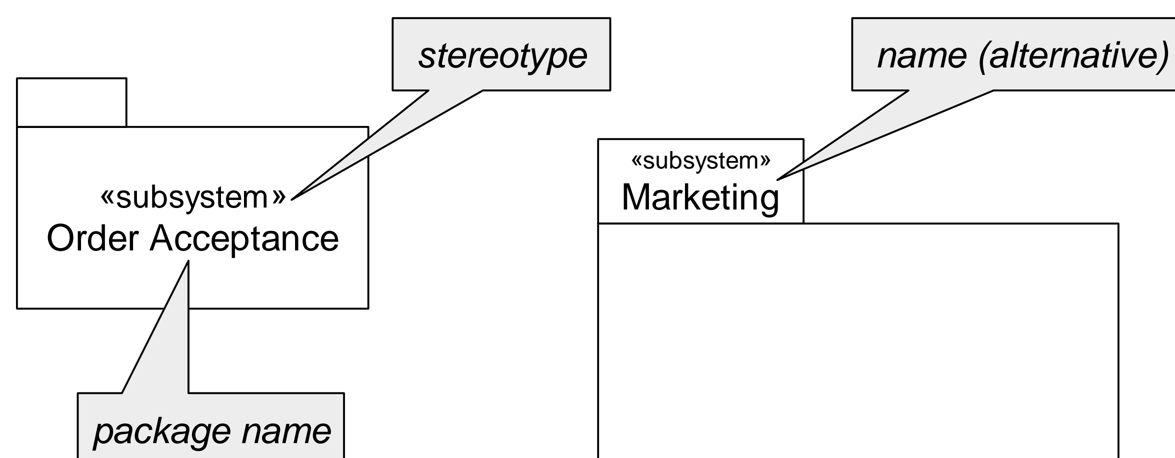

# Packages (1)

A **package** is a general mechanism for the grouping of model elements (e.g., diagrams, classes, ...). Packages can be nested (recursively) in packages. Each element is included in exactly one package (➜ tree).
A package defines a **namespace** for the elements included.

The total system can be regarded as one package.

Special kinds of packages are: **system, subsystem**. Qualification with the stereotype «subsystem».

**Notation**:

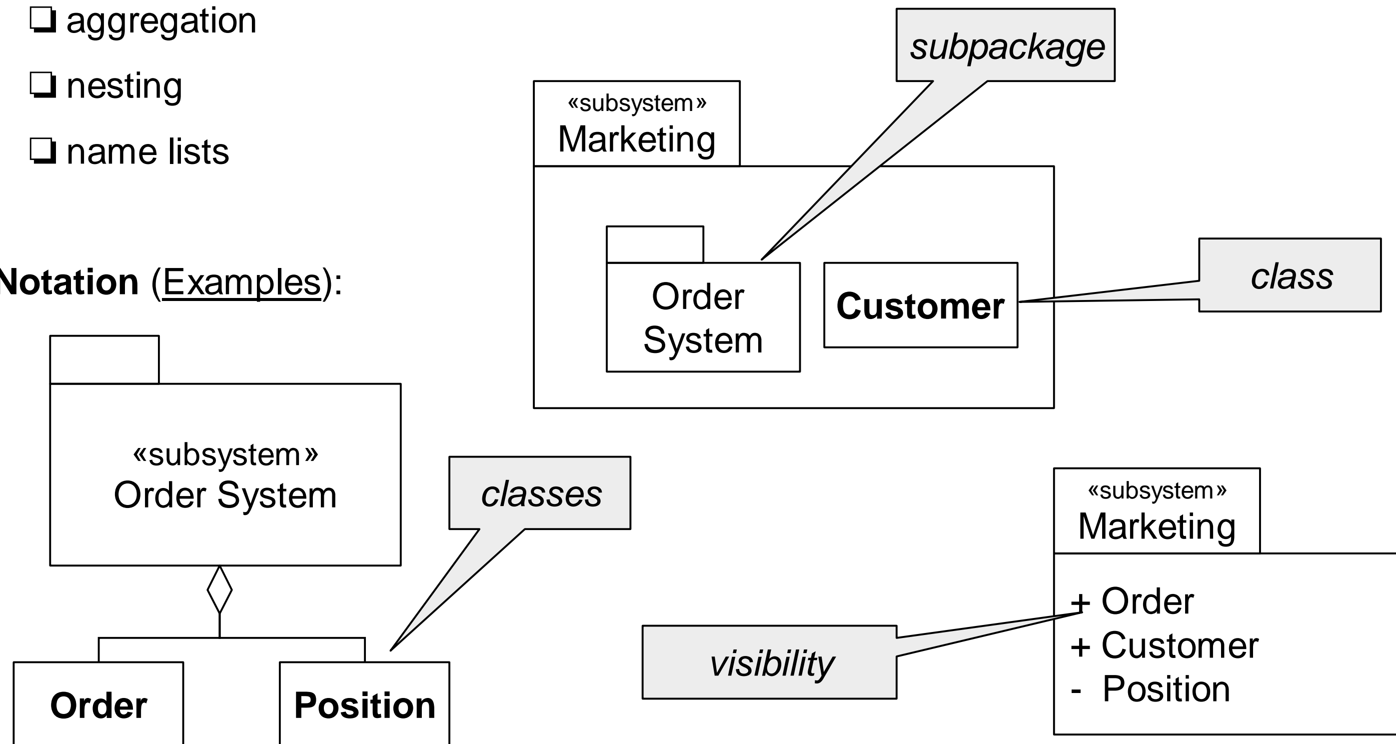


# Packages (2)

Alternatives for the representation of the **content** of packages are:

- ❑ aggregation
- ❑ nesting
- ❑ name lists

**Notation** (Examples):






# Example: Packages in Java

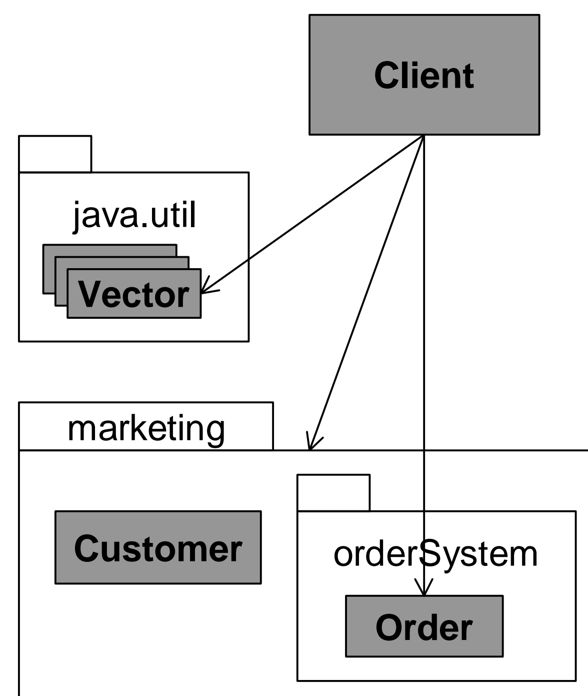
Users (clients) import packages:

```
import marketing.*;
import marketing.orderSystem.Order;
import java.util.Vector;
class Client {...}
```

Package Declaration in Java:

```
package marketing;
public class Customer { ... }

package marketing.orderSystem;
public class Order { ... }
```

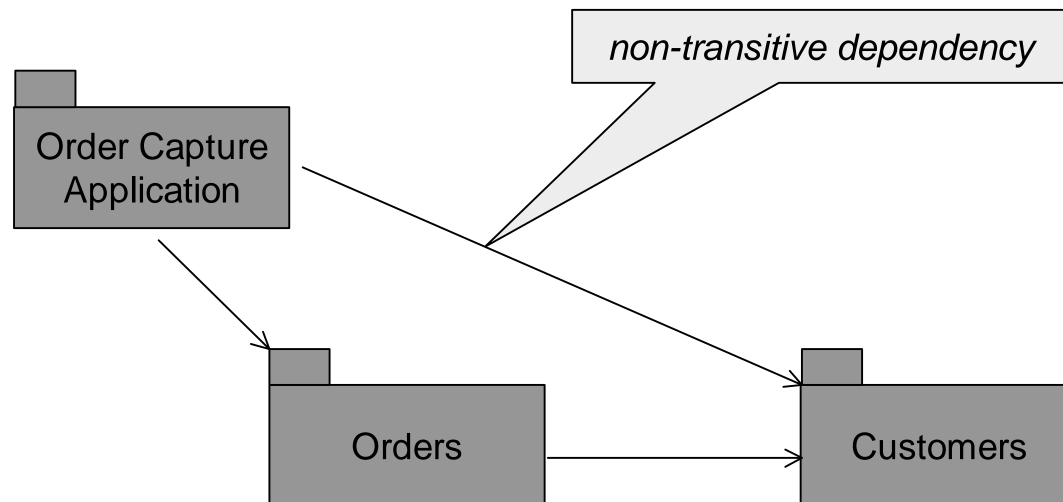


# Package Dependencies

There is an important difference between package dependencies and compilation dependencies:

Dependencies between packages are not transitive.

If transitive dependencies are needed, they have to be modeled explicitly!






# Subsystems

Problem: How do I break down a large system into smaller systems?

| functional decomposition | OO packages |
|---|---|
| Map the total system on functions and subfunctions, starting with the use case. | • Collect classes into subsystems.<br>• Build layers of subsystems.<br>• **Abstraction:** Concentrate on essentials.<br>• **Locality:** Group together related components (data and algorithm).<br>• **Hiding:** Restrict the visibility of details, so that only those parts of a system that need to know the details have access to them. |

# Benefits of Packages and Package Diagrams

Problem: Typically the complexity of a system grows ➡ 100(0) of classes

**Packages allow:**

- ❏ Controlling the complexity:
  - A class diagram should fit on a A4 page.
  - A developer can comprehend about 7 (+/-2) classes at once.
- ❏ Restriction of the namespaces
- ❏ Controlling the propagation of changes
- ❏ Building of nested class hierarchies
- ❏ Building of layered architectures
- ❏ Revelation of dependencies on a higher abstraction level
- ❏ Partitioning of the development in the team
- ❏ Specification of interfaces between groups of classes (subsystems)



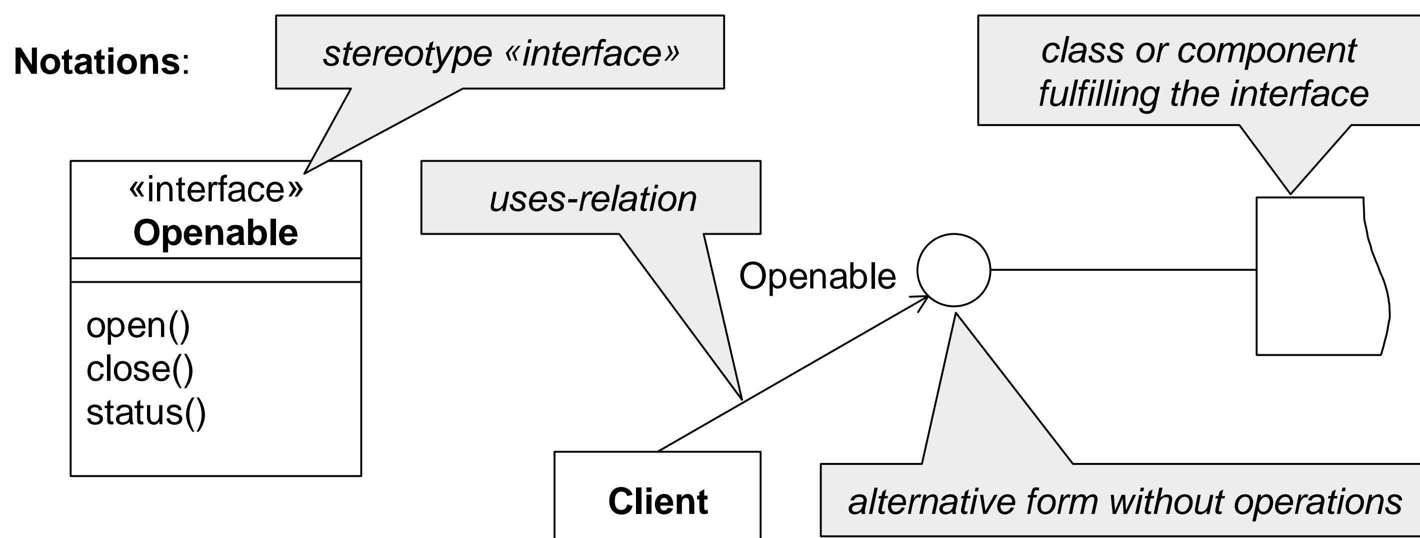
# Interfaces

An **interface** is a named set of operations defining the behavior visible outside, e.g., of a class, component or package.

Interfaces don't have any implementation, no attributes and no associations.

Interfaces can relate to each other in generalization relation.

**Notations**:

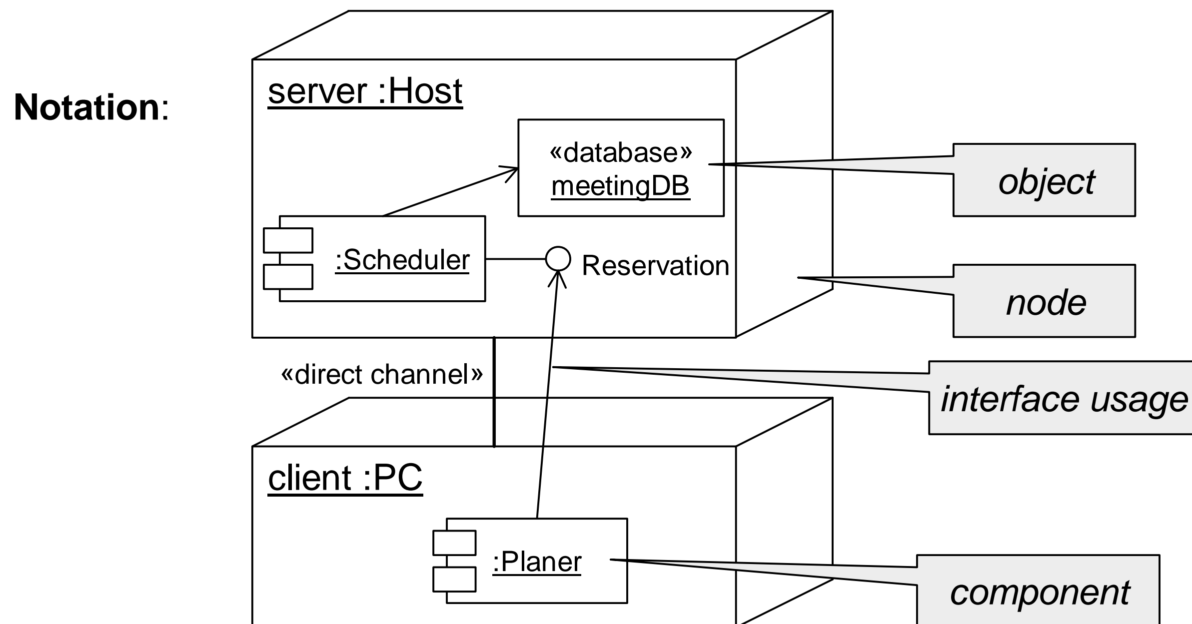


# Deployment Diagram

A **deployment diagram** shows the configuration of a *node* at runtime as well as the *components* (instances) and *objects* residing on it.

Components not existing as runtime object (instance) should appear in component diagrams only .

**Notation**:






# Components (1)

A **component** is a *physical, replaceable* part of a system containing an implementation which conforms to a set of *interfaces* and *realizes* these.

Components have two aspects:

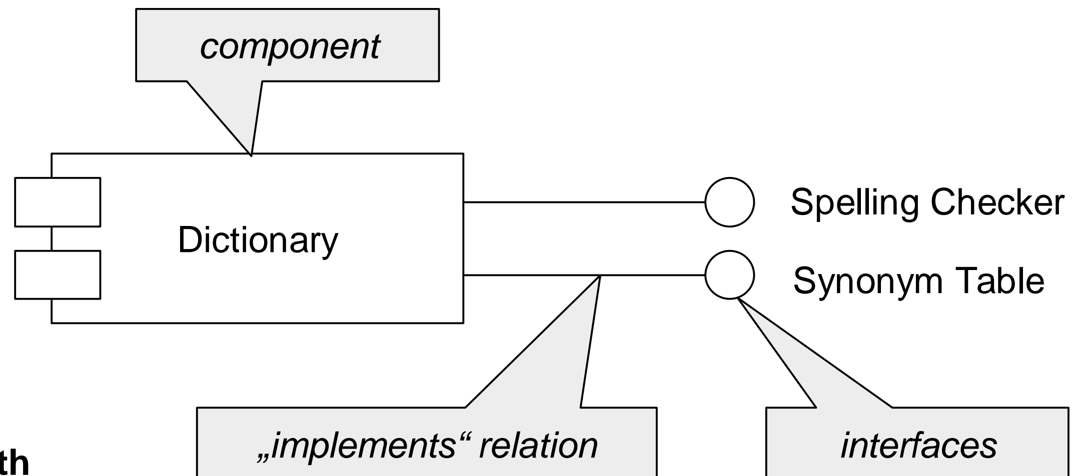
- ❑ **Code**: A component consists of code
  (source code, libraries, executable programs, ...)
  (e.g., *JavaBean*, DLL, EXE-Files, CORBA ORB, ...)
  Components can contain or use components.
- ❑ **Identity:** A component can have identity and state represented by objects. An object which wants to use services of the component must specify the instance.
  (e.g., *Bean* reference, DLL *handle*, process, CORBA-IOR,...)

Example: Spelling checker as component

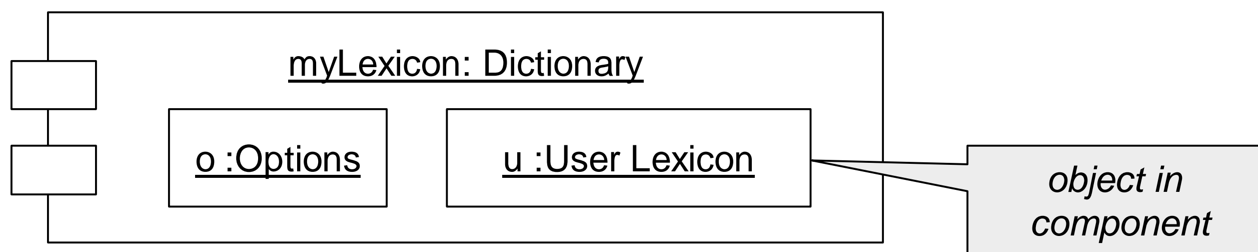
Identity and state by user dictionary, different versions/languages

# Components (2)

**Notation**




**component with identity**



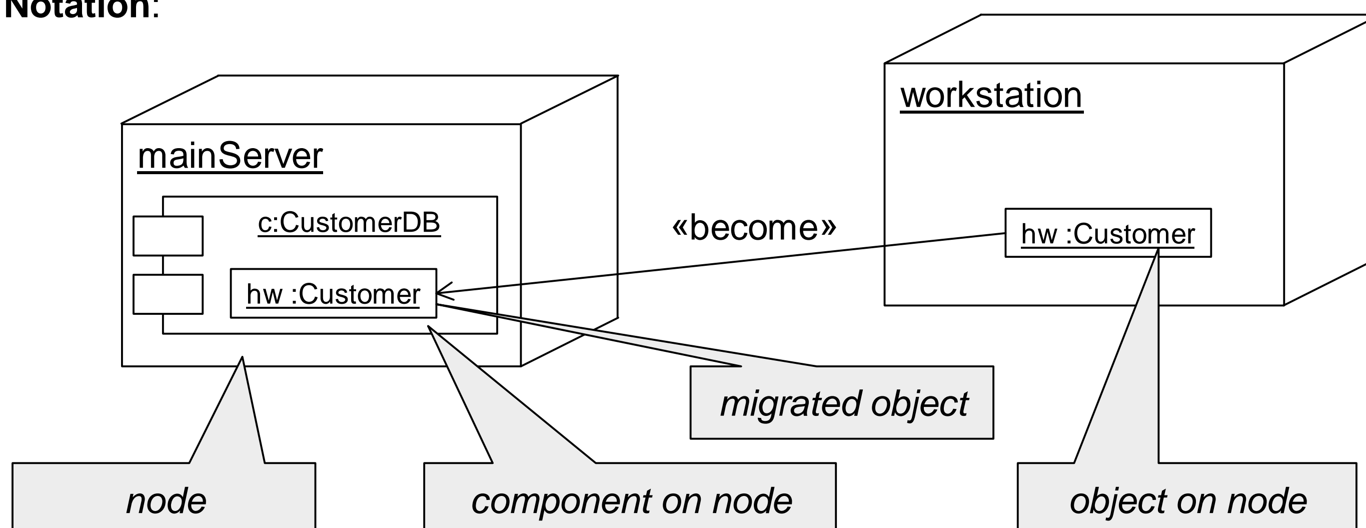



# Node

A **node** is a physical object existing at runtime and representing processing resources which have storage and computation capacity. On a node objects and components can be settled.

Nodes can be computers but also humans or (mechanical) devices or machines (important for business models).

**Notation**:



© J.W. Schmidt, Florian Matthes 1999

# Discussion

- ❑ Components are not trivial because they are conceptual and functional larger than a class. A component unites the behavior and the collaboration of a whole group of classes (➜ collaboration diagram).
- ❑ Independence from other components but mostly collaboration with other components.
- ❑ Components are replaceable with other realizations of the same interface.
- ❑ Components are fundamental parts for the structuring of the architecture.
- ❑ Components can be built recursively. A system can be a component on the next higher inspection level.
- ❑ A component must conform to a set of interfaces. It fulfills the contract which is specified by the interface.
- ❑ Classes (and packages) are **logical abstractions**;
  components represent **physical** things.



# Design Process

**1)** Identify and name the subsystems.

**2)** Constitute visibility for classes within each subsystem.

**3)** Define and cross-check dependencies between packages and the class and interaction diagrams.

**4)** Separate common interfaces in packages, if necessary.