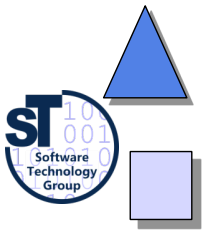
Part II – Black-Box Composition Systems
10. Business Components in a Component-Based Development Process

1. The UML component model
2. Business component model of the Cheesman/ Daniels process
3. Identifying business components



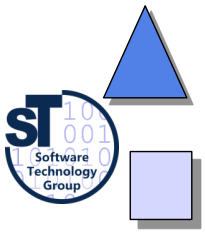
Prof. Dr. Uwe Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
http://st.inf.tu-dresden.de
12-0.2, 09.04.12

Literature

- J. Cheesman, J. Daniels. UML Components. Addison-Wesley.

10.1 Big Objects and The UML Component Model

(Cheesman-Daniels)



Natural and Dependent Types

- An object with a **natural type (entity type)** lives on its own and exists independent of context and collaborators
  - The type does not depend on other types (**independent type**)
    - Hotel vs. HotelRoom
    - Car vs. Screw or Motor
  - Types that depend on others are called **dependent types.**

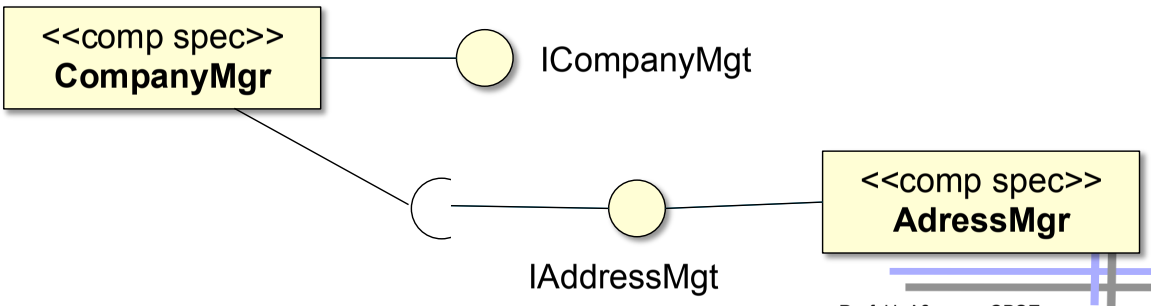
Big Objects (Bobs)

- A **big object (bob)** is complex, hierarchical object with a natural type

Component Specification with UML Components

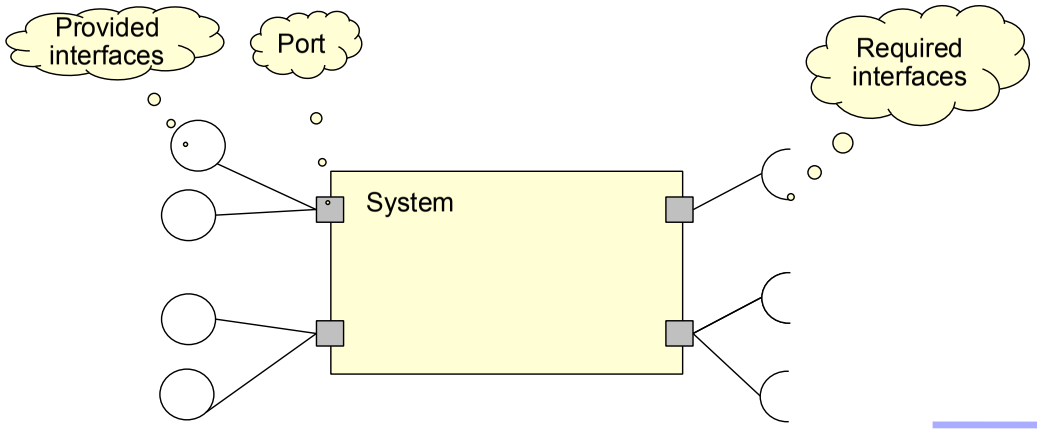
- A **UML component** is a hierarchical class (classifier, type) with *provided* and *required* interfaces (roles)
  - Provided interfaces (roles) use „lollipop“ notation
  - Required interfaces (roles) use „plug“ notation

- Some components are required to use specific other interfaces



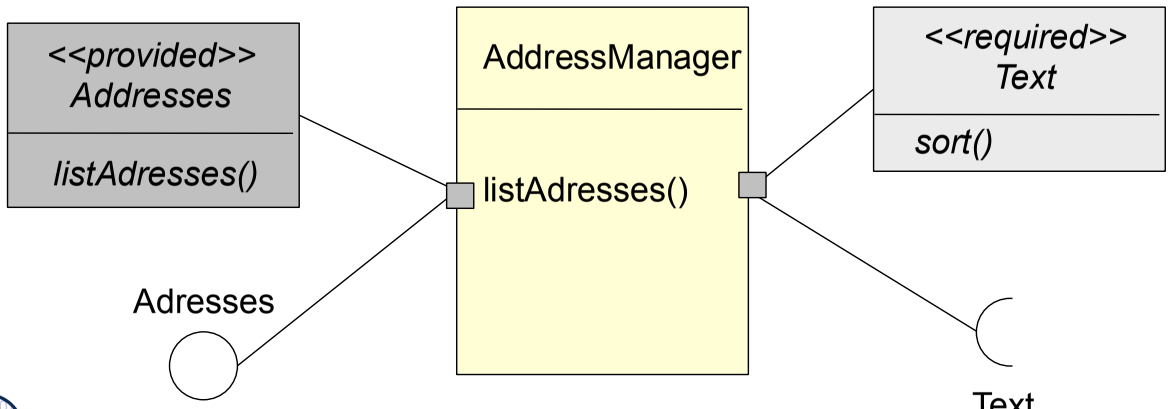
Ports

- A **port** is a connection point of a UML component.
  - A port has a set of roles (interfaces)
  - It may be represented by a **port object (gate)**



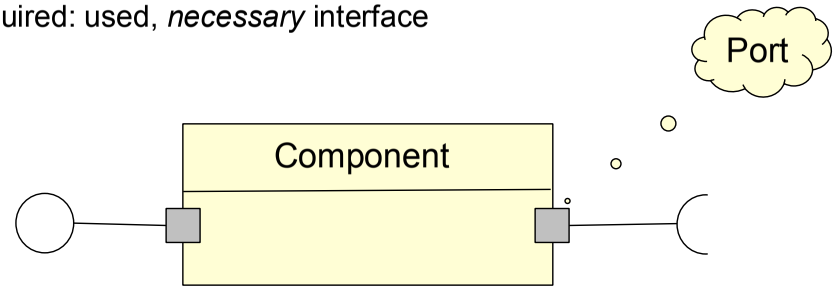
Lollipops und Plugs (Balls and Sockets)

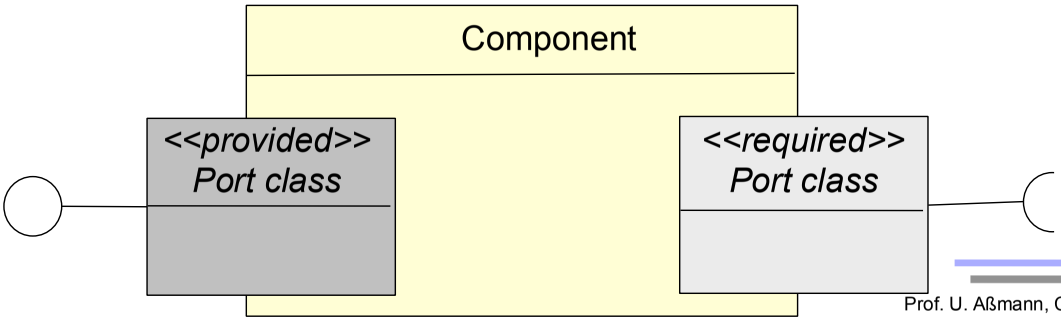
- For a UML component, *provided* and *required interfaces* can be distinguished
  - A required interface specifies what the current class needs to execute.



Ports

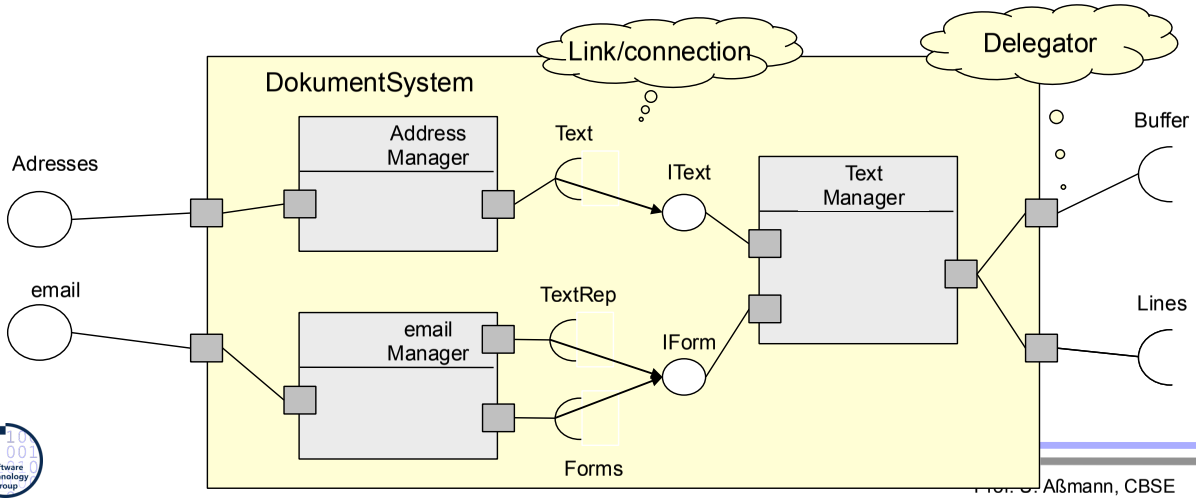
- Ports consist of **port classes** with interfaces and behavior in form of **interface automata**
  - provided: normal, *offered* interface
  - required: used, *necessary* interface





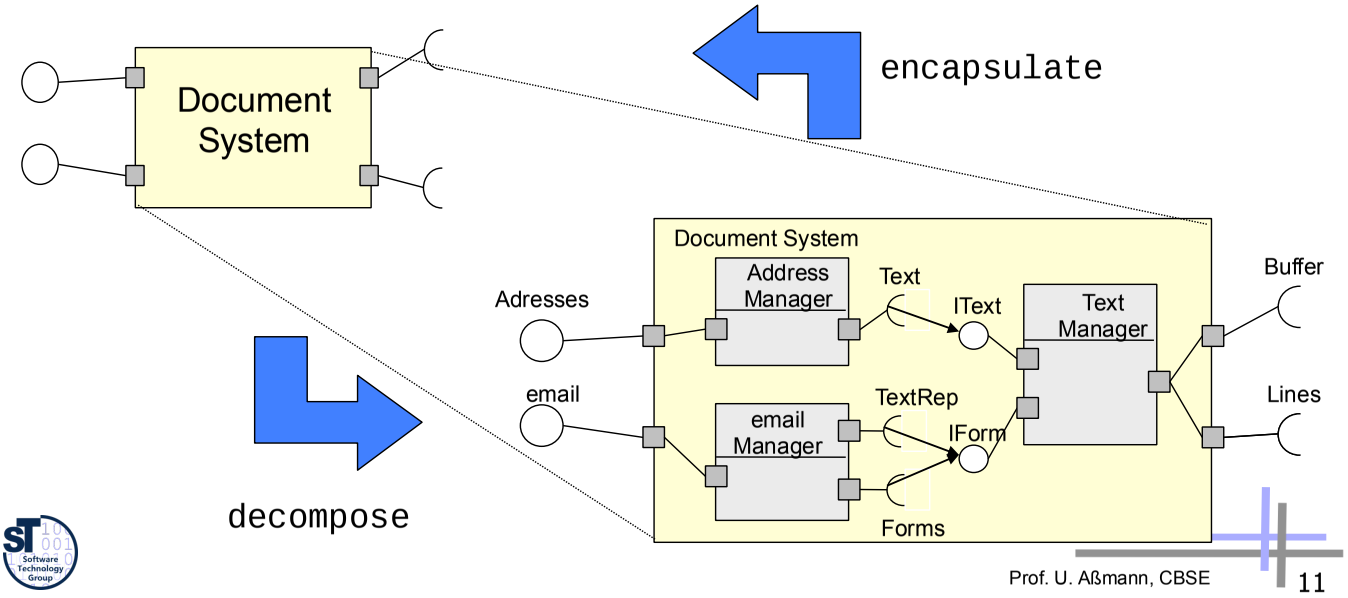
Nesting of UML Components

- UML components
  - Ports are connected by *links (connections)*
  - *Delegation link*: links outer and inner port



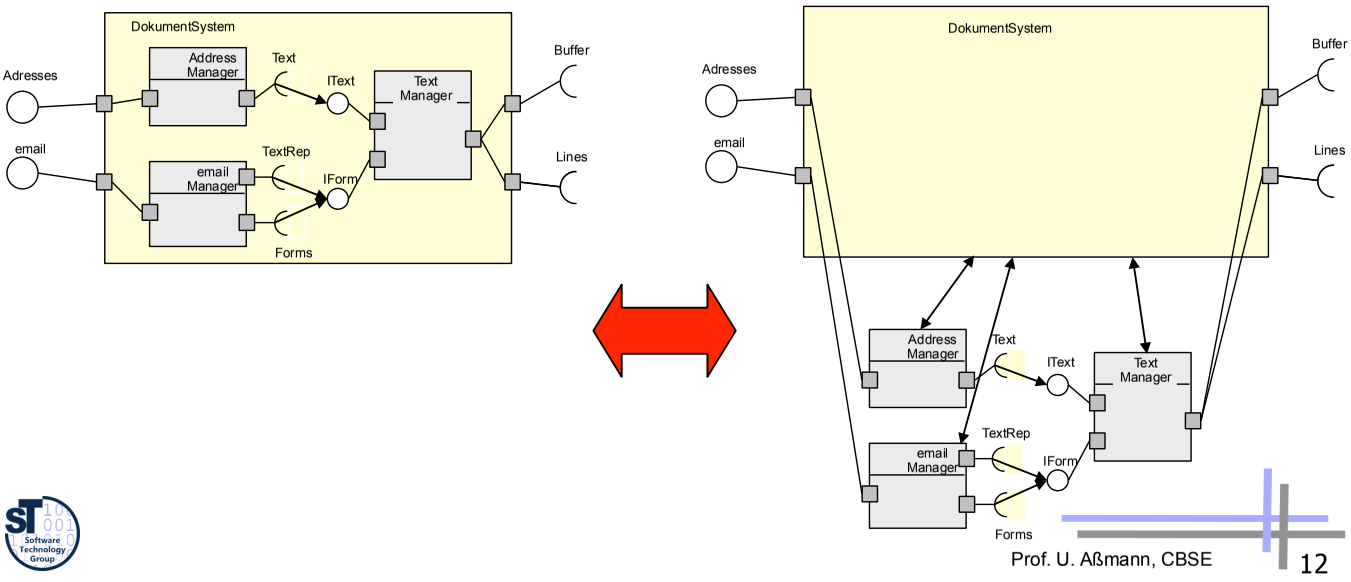
Refinement of UML Components

- UML components are nested, i.e., are bobs.
- Nesting is indicated by *aggregation* and *part-of relationship*.
- Nesting is introduced by an encapsulation operator `encapsulate`.



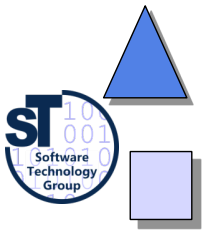
Encapsulation means Aggregation

- *Nesting* means *Aggregation*
  - A UML component is a package and a façade for all subcomponents



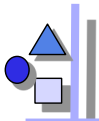
10.2 A Business Component Model

(Cheesman-Daniels)



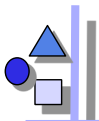
Goals of the Cheesman-Daniels Process

- The Cheesman-Daniels Process identifies big components in UML class diagrams
  - It bridges *domain modelling* with *use case modelling* (functional requirements)
- Steps:
  - Find out business objects (big objects) of application
  - Group business objects to components for change-oriented design and reuse
  - Specify contracts for the components
- Be aware: the Cheesman-Daniels Process can be employed also for many other component models of this course, such as
  - Black box component models, such as EJB, Corba, .NET
  - Grey-box component models:
    - Generics (e.g., class diagram templates)
    - Fragment component models (e.g., advice groups in aspects)
    - Class-role models



Business Objects are Complex Objects

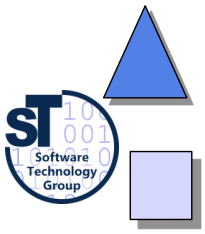
- A **business object (domain object)** is a bob with a natural type of the domain model (business model)
- Usually, business objects (domain objects) are large hierarchical objects
  - They can consist of thousands of smaller objects of dependent types (part-of relation)
  - They can play many *roles* with *context-based types*



Business Component Model

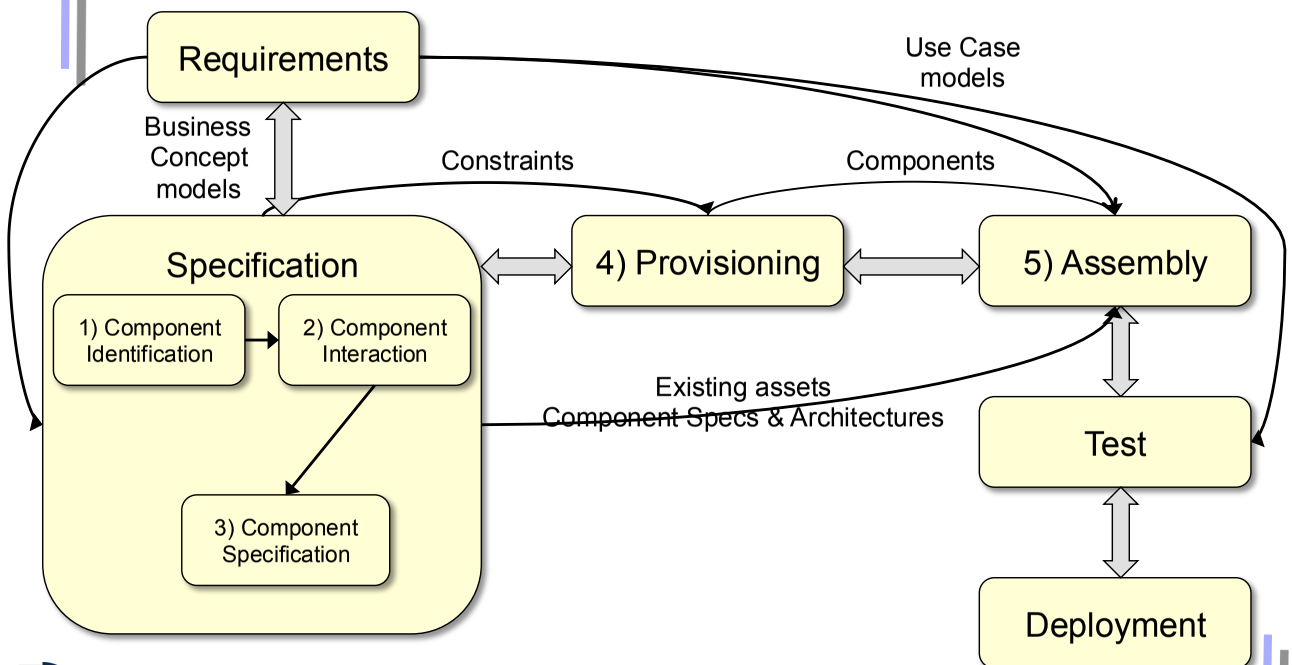
- In the Cheesman-Daniels component model, a **business component** consists of a set of business objects and other business components (part-of relation)
- The smallest component is a *business object*
  - groups several interfaces together.
  - has several provided interfaces
  - has several requried interfaces
    - The business objects are the logical entities of an application
    - Their interfaces are re-grouped on system components for good information hiding and change-oriented design
  - Has a specification containing all interfaces and contracts
  - Has an implementation
  - UML-CD are used (UML profile with stereotypes)

10.3. Identifying Business Components



Identifying Business Components with the Cheesman-Daniels Process

- Overall development process

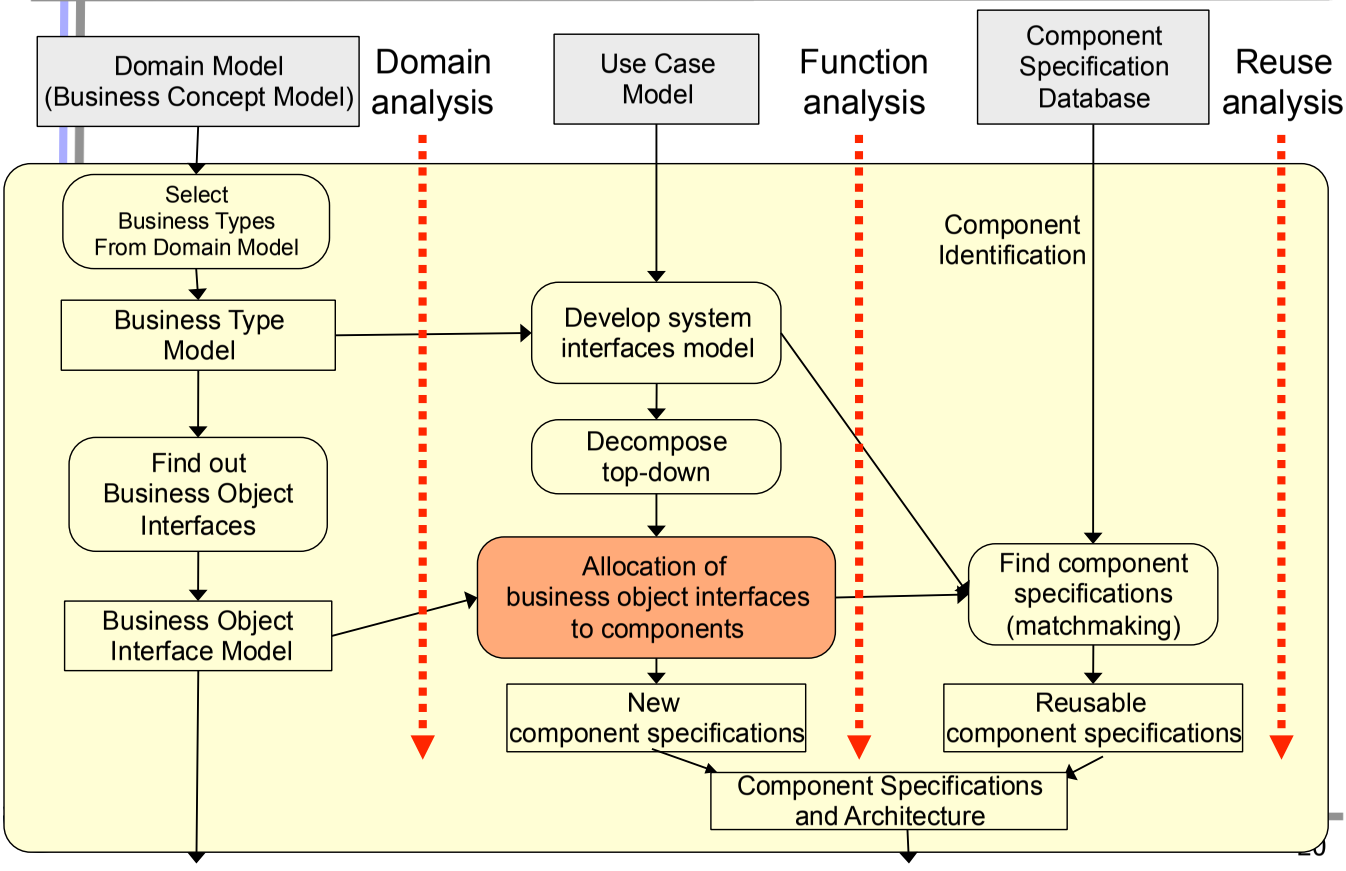


Simplified version of Fig. 2.1 from Cheesman/Daniels

Artifacts of the Cheesman/Daniels Process

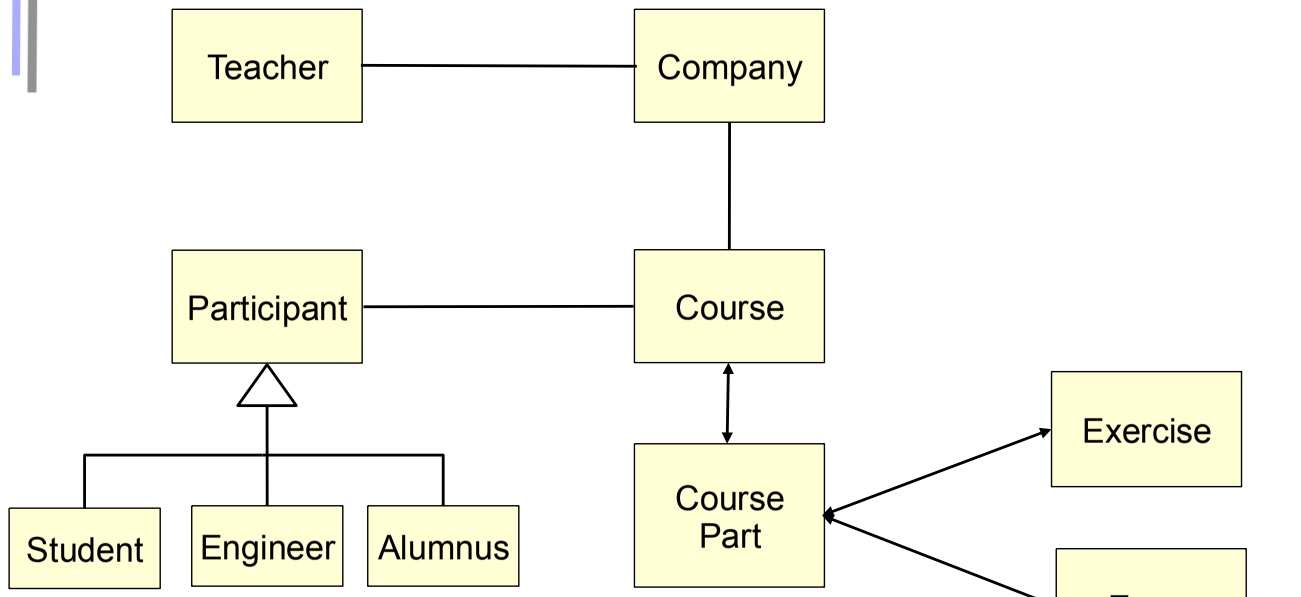
- Requirement artifacts:
  - *Business concept model (business model, domain model):* describes the business domain (application domain)
  - *Use case model* (requirements model)
- System artifacts, derived from the business concept model:
  - *Business type model*, derived from domain model.
    - Represents the system's perspective on the outer world (more attributes, refined class structures from the system's perspective)
  - *Business object interface model*, containing the business objects and all their interfaces
  - *Business object model*, derived from the business object interface model by adding operations
- System component artifacts
  - Component interface specifications: one contract with the client
  - Component interface information model (state-based model)
  - Component specifications: all interface specifications of a component plus constraints.
  - Component architecture: wiring (topology) of a component net.

10.3.1 Component Identification (Step 1)



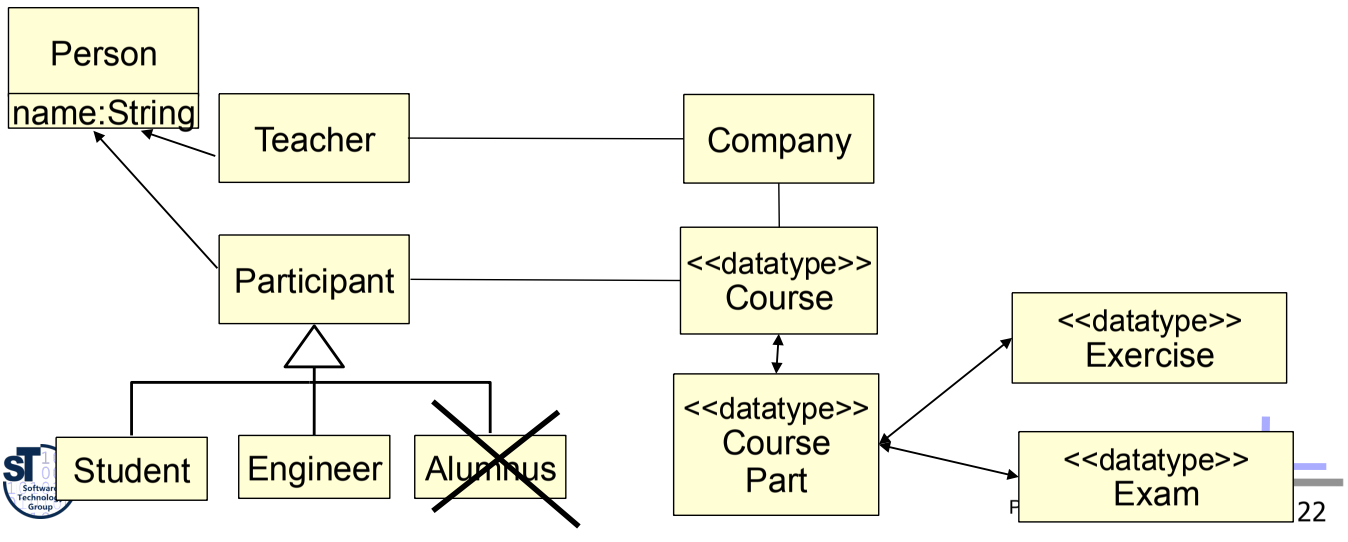
Ex.: Domain Model of a Course-Management System

- Collects all concepts of the domain (aka business concept model)



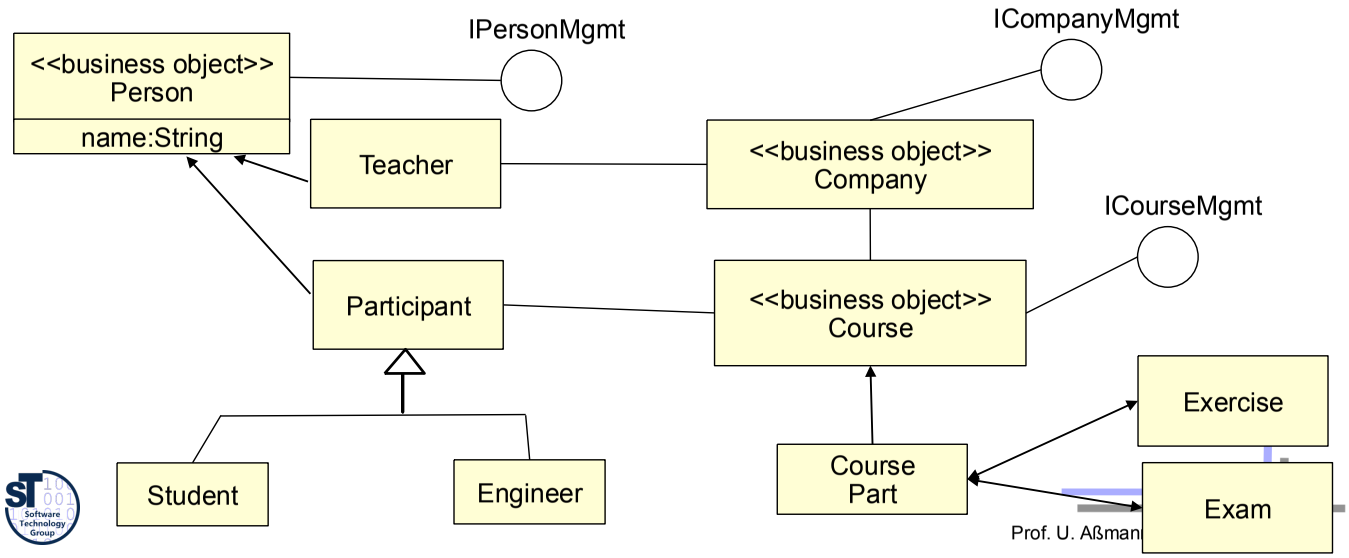
Business Type Model

- Defines system types from the domain model
  - Eliminates superfluous concepts
  - Adds more details
  - Distinguish datatypes (passive objects)



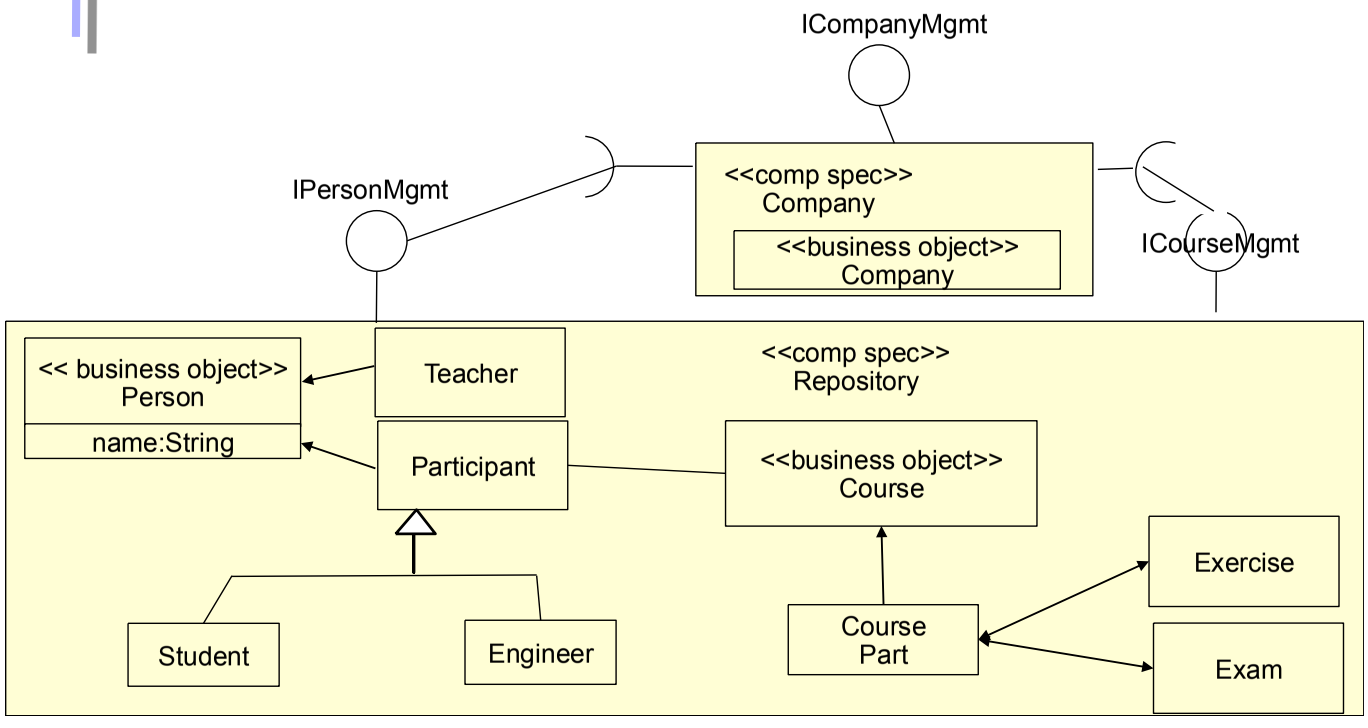
Business Object Interface Model

- Identifies business objects from the business type model
  - And defines *management interfaces* for them
  - Here, only Company, Course, Person are business objects, all others are dependent types



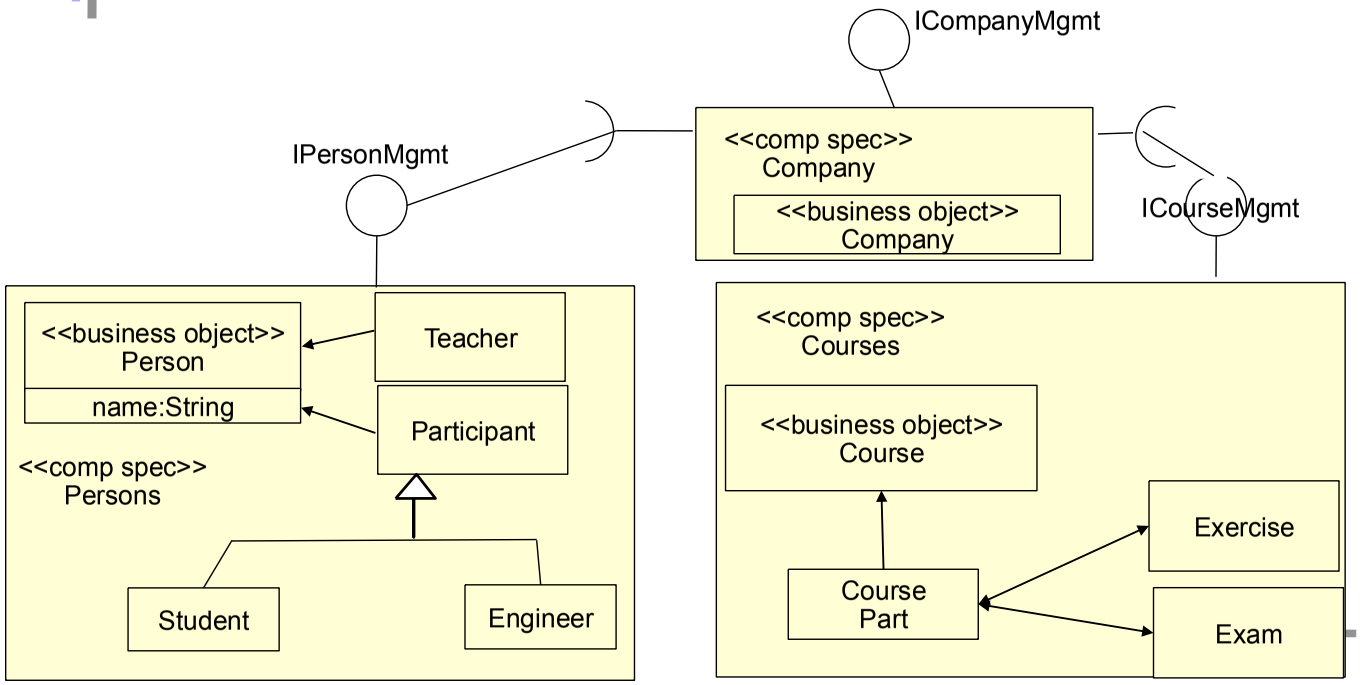
Component Identification (Version 0.1)

- Group classes and interfaces into reusable components



Alternative Component Identification (0.1)

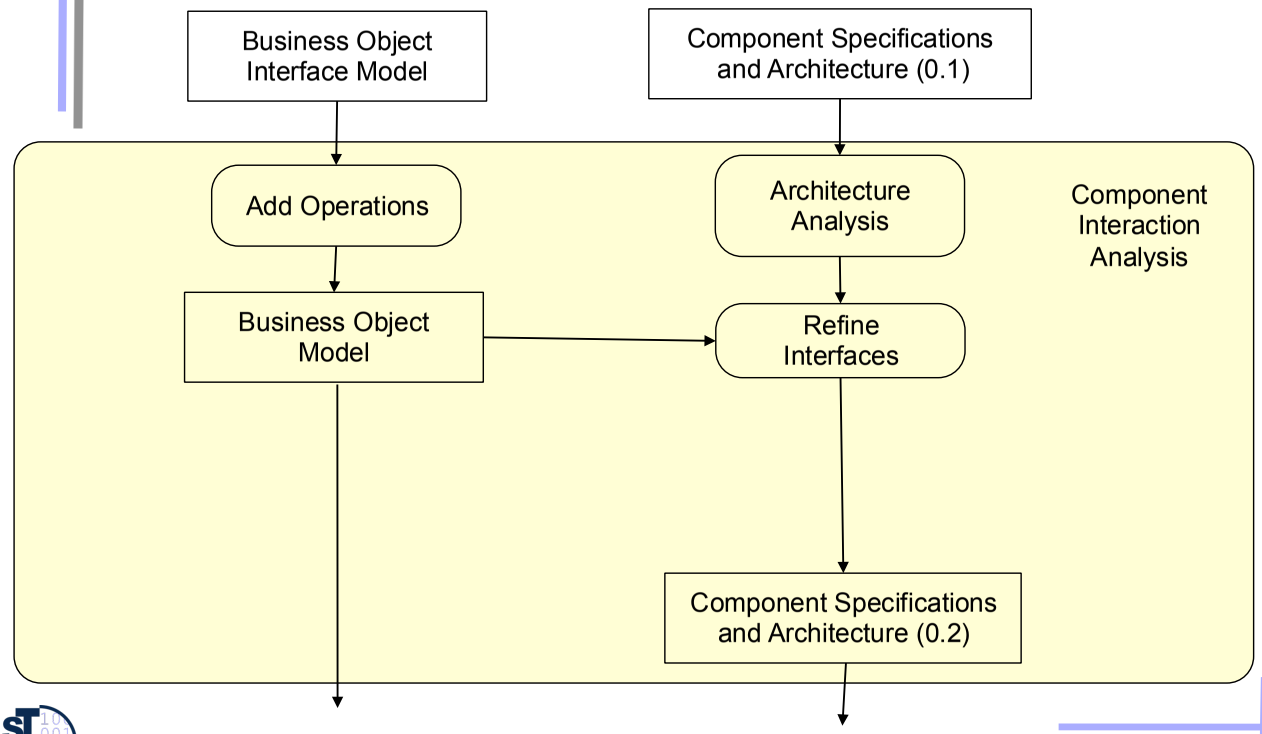
- Group classes and interfaces into components
- Person management might be reuseable



Component Identification

- The **component identification** subprocess attempts to
  - Create a business object interface model from the domain model (still without methods)
  - Attempts to group these interfaces to initial *system component specifications*
    - The grouping is done according to
      - *information hiding*: what should a component hide, so that it can easily be exchanged and the system can evolve?
      - *Reuse considerations:* which specifications of components are found in the component specification repository, so that they can be reused?
- There is a tension between business concepts, coming from the business domain (problem domain), and system components (solution domain). This gap should be bridged.

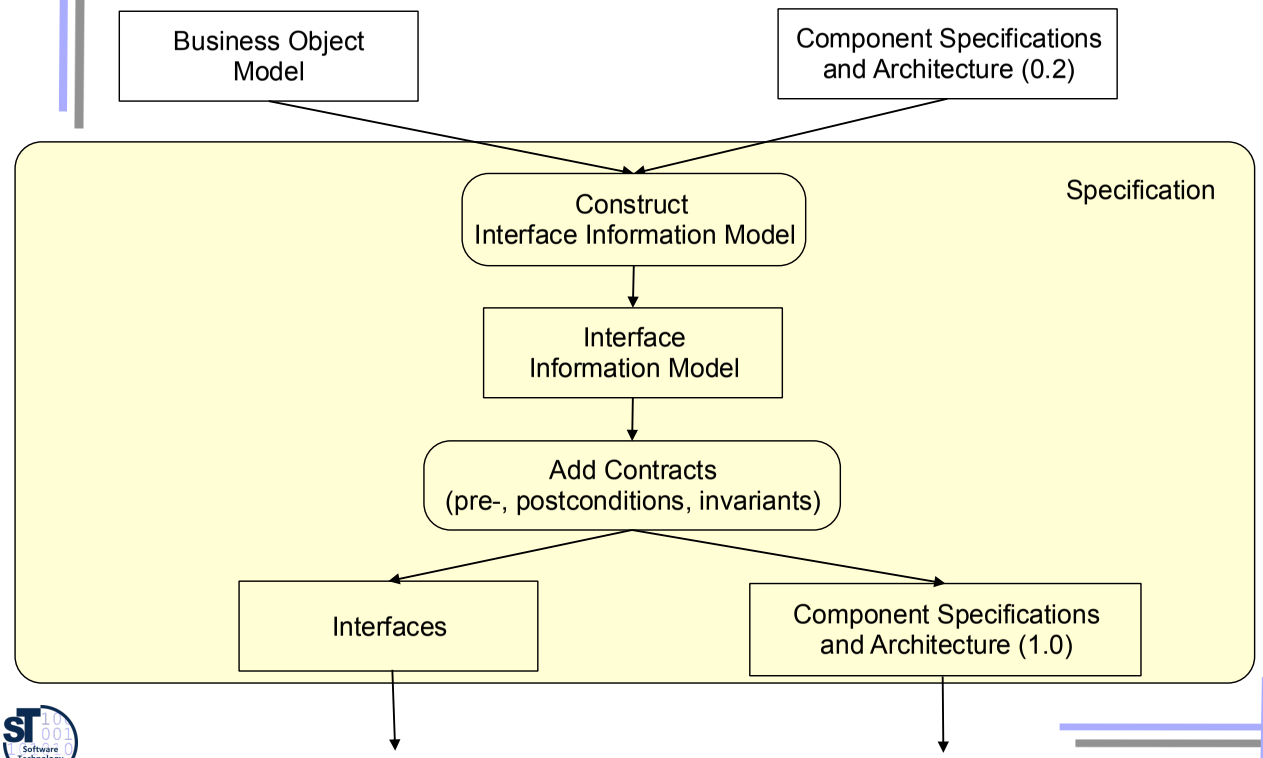
10.3.2 Component Interaction Analysis (Step 2)



Component Interaction Analysis

- Is basically a refinement of the first stage
  - Removing,
  - Regrouping,
  - Augmenting,
  - Producing component specifications and wirings in a version 0.2
- Additionally, operations are added to business object interfaces
  - And mapped to internal types.

10.3.3 Component Specification (Step 3)



Component Specification (Step 3)

- Specification of declarative contracts for UML components in OCL
- Invariant construction:
  - Evaluate business domain rules and integrity constraints
  - Example:

```
context r: Course
-- a course can only be booked if it has been allocated in
   the company
inv:  r.bookable = r.allocation->notEmpty
```

- Pre/Postconditions for operations
  - Can only be run on some state-based representation of the component
  - Hence, the component must be modeled in an *interface information model*
  - Or: be translated to implementation code (e.g. Java using an OCL2Java Compiler)

10.3.4. Provisioning (Realization, Implementation) (Step 4)

- Provisioning selects component implementations for the specifications
  - Choosing a concrete implementation platform (EJB, CORBA, COM+, ...)
  - Look up component implementations in implementation repositories
    - Write adapters if they don't fit exactly
  - Program missing components
  - Store component implementations and specifications in database for future reuse

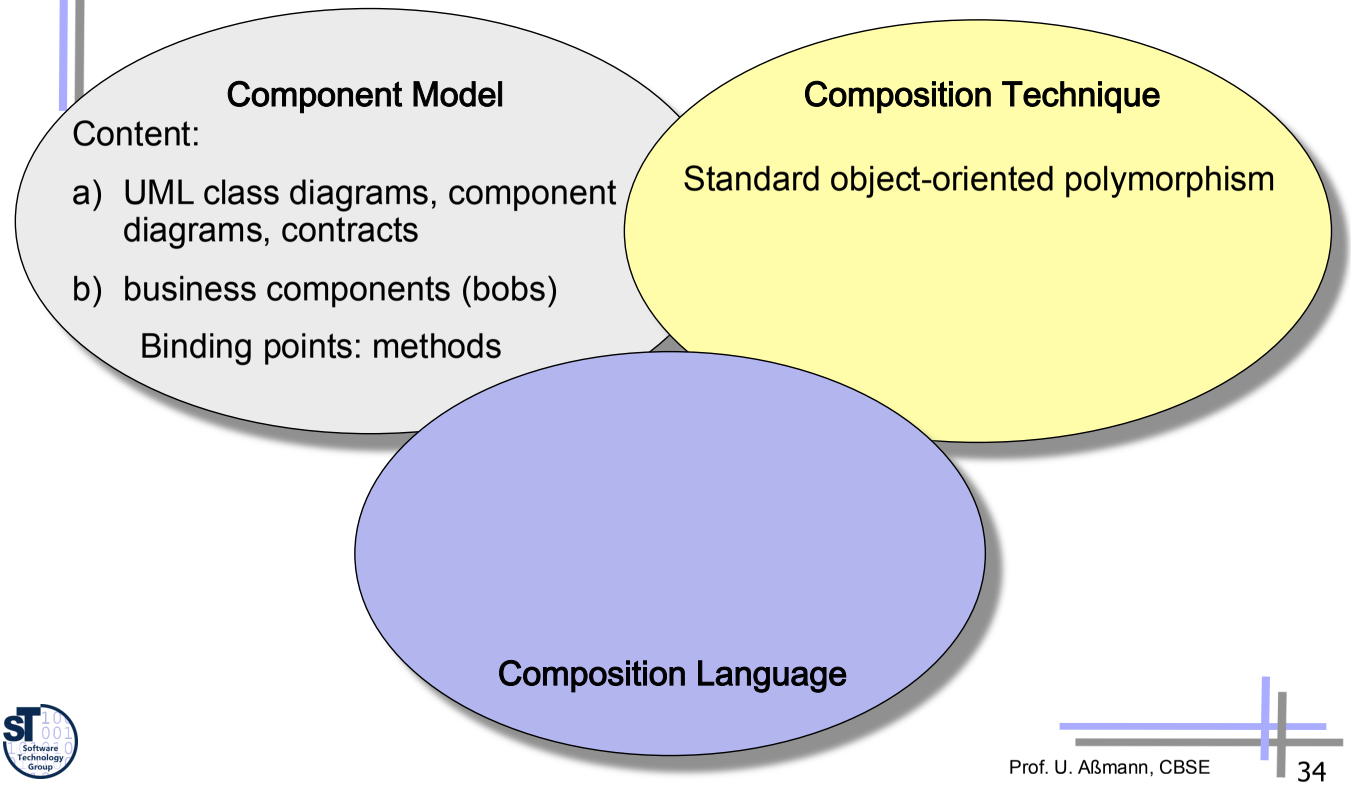
10.3.5 Assembly (Step 5)

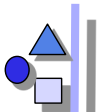
- Puts together architecture, component specifications and implementations, existing components
  - We will see more in the next lectures

Weaknesses

- No top-down decomposition of components
  - part-of relationship is not really supported
- Reuse of components is attempted, but
  - Finding components is not supported (see companion lecture)
    - Metadata
    - Facet-based classification

Cheesman-Daniels' Business Component Model as Composition System





The End