Methodische und Praktische Grundlagen der Informatik (MPGI 3) WS 2008/09

Softwaretechnik

Steffen Helke

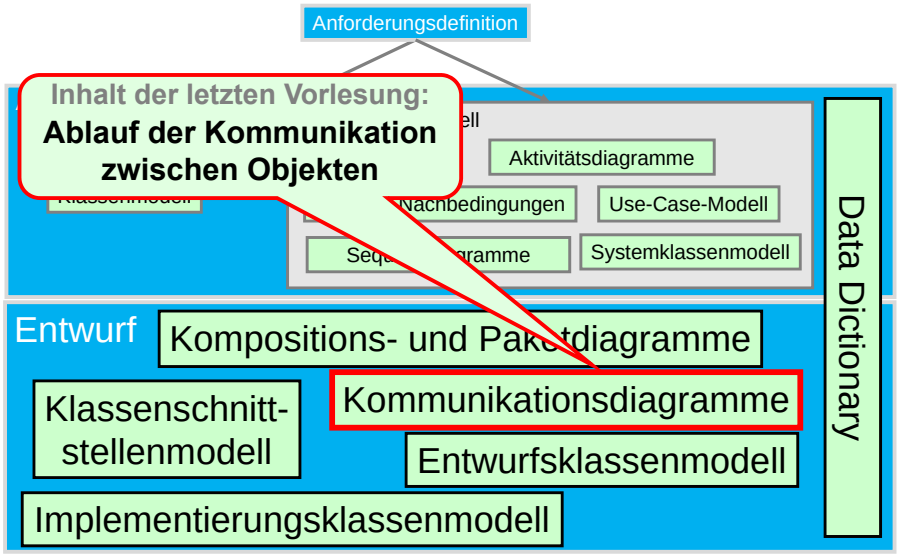
Andreas Mertgen (Organisation)

Rojahn Ahmadi, Georgy Dobrev, Daniel Gómez Esperón, Simon Rauterberg, Jennifer Ulrich (Tutoren)

Was machen wir heute?

- **Rückblick**
  - Kommunikationsdiagramme
- **vom Entwurf zum Code**
  - **Implementierungsklassenmodell**
  - Festlegen von **Klassenschnittstellen**
  - **Entwurfsmuster**

Entwurfsmodelle in der UML



Zusammenfassung: Vorgehen im Entwurf

1. **Festlegen der Systemarchitektur**
   - Kompositions- oder Paketdiagramme
2. **Klassenentwurf**
   - Entwurfsklassenmodell
3. **Spezifikation von Schnittstellen**
   - Kommunikations- und Sequenzdiagramme
4. **Detailentwurf**
   - Implementierungsklassenmodell, Klassenschnittstellenmodell

Aufstellen von Kommunikationsdiagrammen

1. Bestimmen der Eingaben
- aus dem Operationsschema

2. Festlegen des Control-Objekts
- Kommunikation zwischen Boundary und Controller nicht behandeln

3. Behandeln von Fallunterscheidungen
- Identifizieren beteiligter Objekte
- Kommunizieren, Verarbeiten und Speichern von Daten
- Erzeugen und Löschen von Objekten
- Versenden von Rückmeldungen an Akteure

4. Prüfen auf Konformität zum Systemklassenmodell
- Aufbau eines Entwurfsklassenmodells

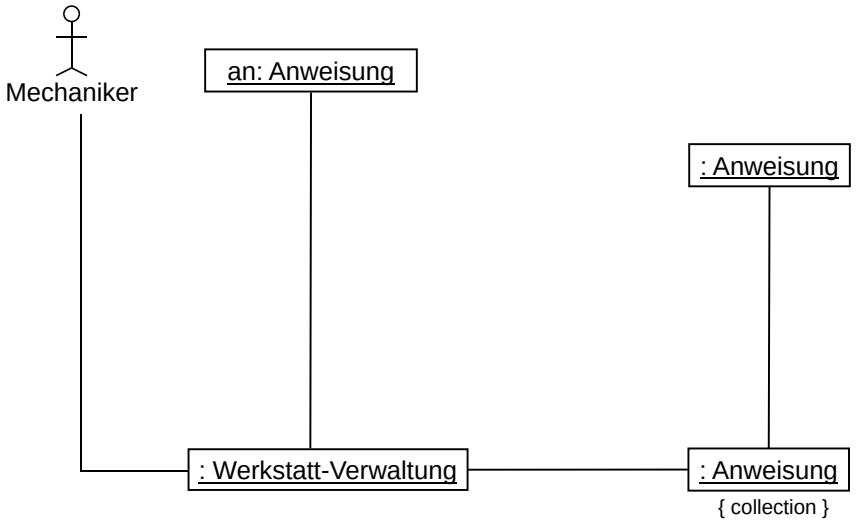
Notationen für Kommunikationsdiagramme

: Werkstatt-Verwaltung — : Anweisung	**Struktureller Aufbau:** Objekte und Links
: Werkstatt-Verwaltung — 1: get_status → : Anweisung	**Dynamische Ablauf:** Annotierung der Links mit gerichteten synchronen Nachrichtenflüssen
: Werkstatt-Verwaltung — 1: suche(aid) → : Anweisung { collection }	**Collections:** Verwaltung mehrerer Objekte
: Anweisung — 3: **create**(anz) → : Material-Posten	**Erzeugen/Löschen:** Objektdynamik mit **create** oder **destroy**
: Anweisung — 3 **[B]**: create(anz) → : Material-Posten	**Conditions:** Bedingte Kontrollflüsse
: Werkstatt-Verwaltung — 1: **an**=finde(aid) → : Anweisung { collection }; 2: setze_status; :Auftrag; **? an:**	**Variablen:** lokaler Geltungsbereich
: Werkstatt-Verwaltung — **1**: an=finde(aid) → : Anweisung { collection }; : Anweisung — **1.1**: getid	**Nummerierung :** Verschachtelte Kontrollfüsse (Delegation)

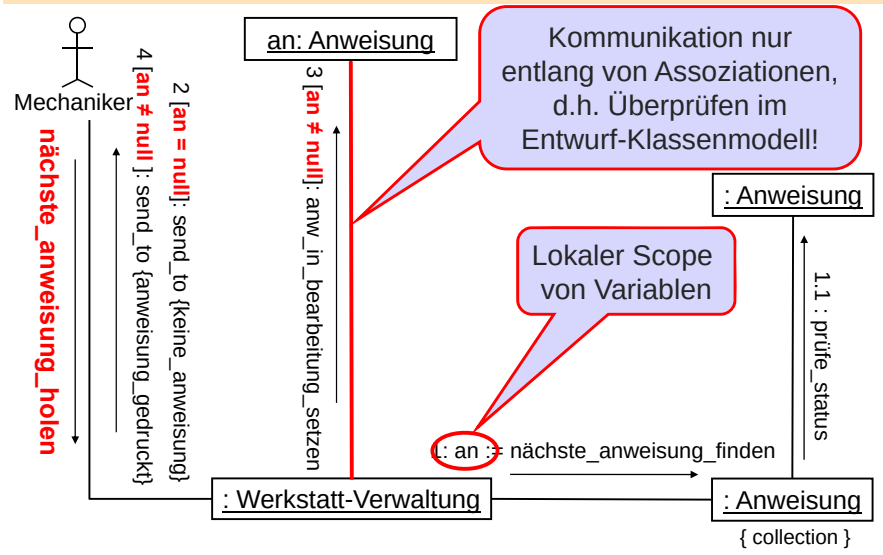
Einfaches Beispiel aus der letzten Vorlesung

nächste_anweisung_holen

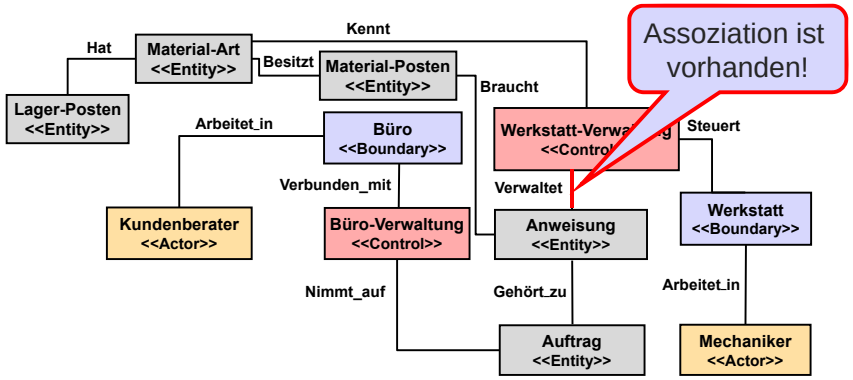
Umsetzung: Struktureller Aufbau
Beispiel: nächste_anweisung_holen



Umsetzung: Dynamischer Ablauf
Beispiel: nächste_anweisung_holen



Abgleich von Kommunikationsdiagramm und Entwurfsklassenmodell

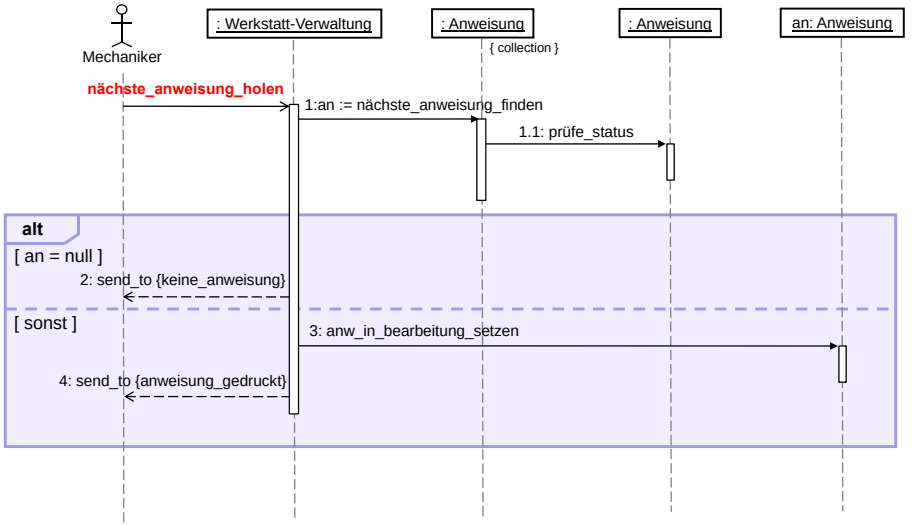


Sollte keine Assoziation vorhanden sein, so ist diese nachträglich hinzuzufügen!

Unnötige Assoziationen sind im Entwurfsklassenmodell zu entfernen!

Ein Kommunikationsdiagramm kann auch durch ein Sequenzdiagramm ausgedrückt werden.

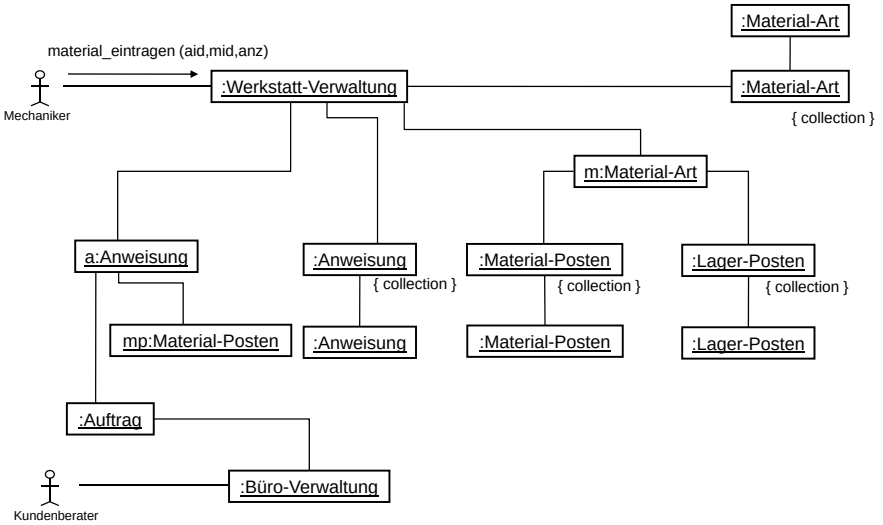
Komunikation in Sequenzdiagramm: nächste_anweisung_holen



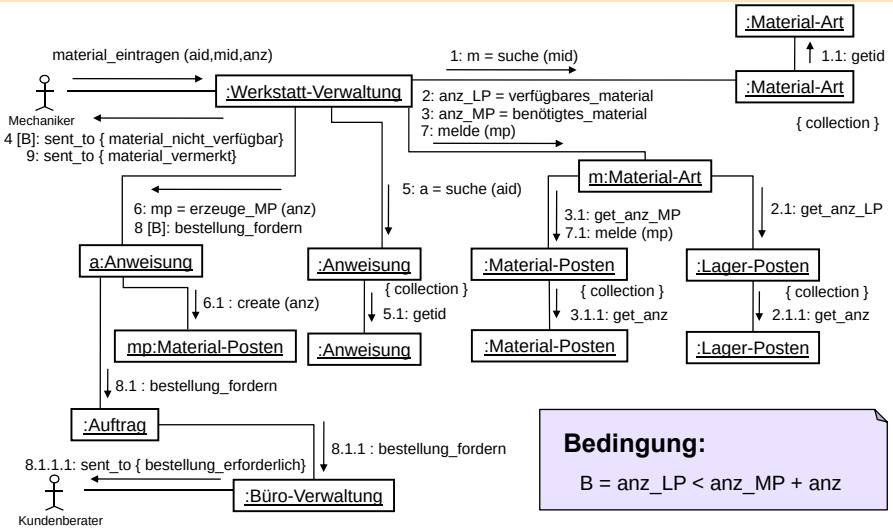
Komplexeres Beispiel aus der letzten Vorlesung

material_eintragen

Umsetzung: Struktureller Aufbau
Beispiel: material_eintragen



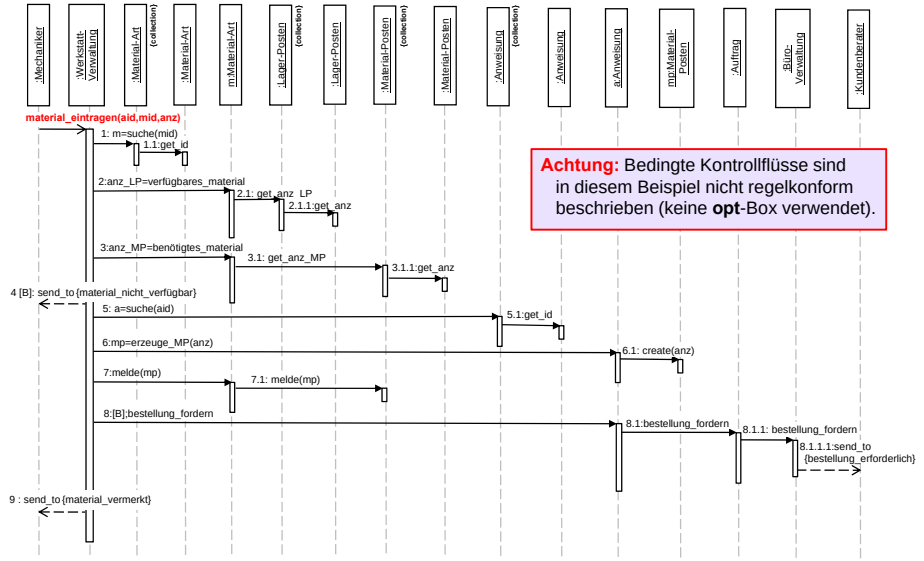
Umsetzung: Dynamischer Ablauf
Beispiel: material_eintragen



Bedingung:

B = anz_LP < anz_MP + anz

Kommunikation als Sequenzdiagramm: material_eintragen



Achtung: Bedingte Kontrollflüsse sind in diesem Beispiel nicht regelkonform beschrieben (keine **opt**-Box verwendet).

Präzisieren der Klassendiagramme

1. Festlegen der Systemarchitektur

Kompositions- oder Paketdiagramme

2. Klassenentwurf

Entwurfsklassenmodell

3. Spezifikation von Schnittstellen

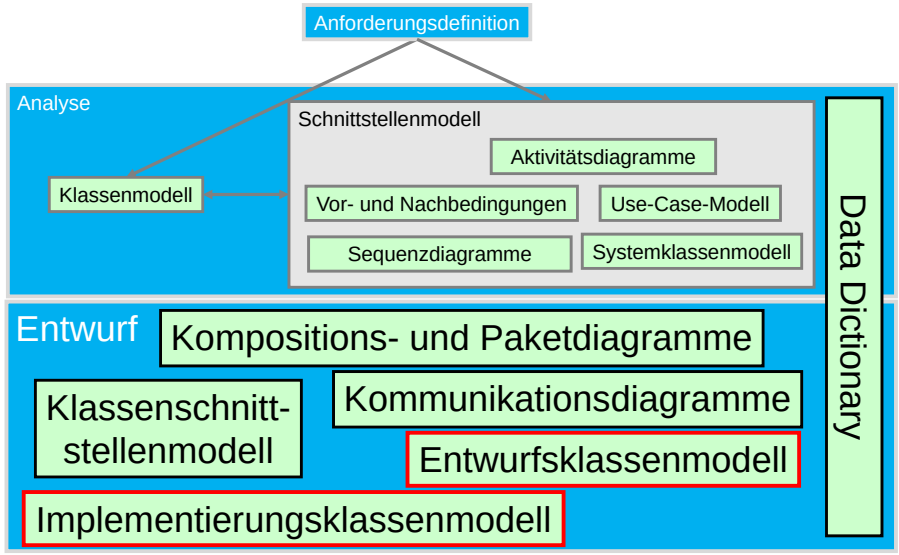
Kommunikations- und Sequenzdiagramme

4. Detailentwurf

Implementierungsklassenmodell,

Klassenschnittstellenmodell

Entwurfsmodelle in der UML



Präzisierungen in Klassendiagrammen

1. Entwurfsklassenmodell

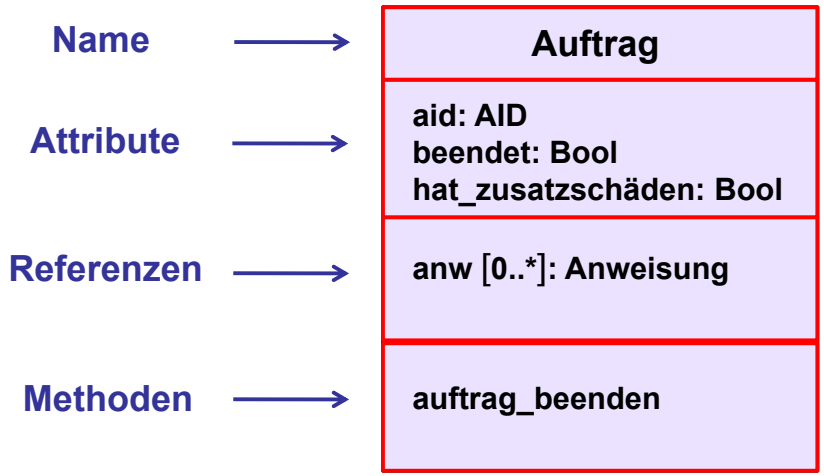
- **vollständige** und detaillierte **Beschreibung von Attributen und Assoziationen**

2. Implementierungsklassenmodell

- Beschreibe alle Klassenelemente mit **Sichtbarkeiten von Attributen und Methoden** und detaillierten Datentypen
- Hinzufügen von **Referenzen** (gerichtete Assoziationen)

Verfeinerung von Klassen

Beispiel: Klasse Auftrag



Erweiterte Syntax für Attribute
Sichtbarkeit (visibility) wie in C++ oder Java

Zeichen	Bedeutung	Umschreibung der Sichtbarkeit
+	public	für alle
#	protected	innerhalb der Klasse oder in einer ihrer Unterklassen
~	package	innerhalb des Pakets der Klasse
-	private	nur innerhalb der eigenen Klasse

Erweiterte Syntax für Attribute
Beispiel für Sichtbarkeit in der Klasse Auftrag

Wir empfehlen Attribute als privat zu deklarieren. Lese- und Schreibzugriffe durch andere Objekte sollten besser über Methoden (get/set) realisiert werden.

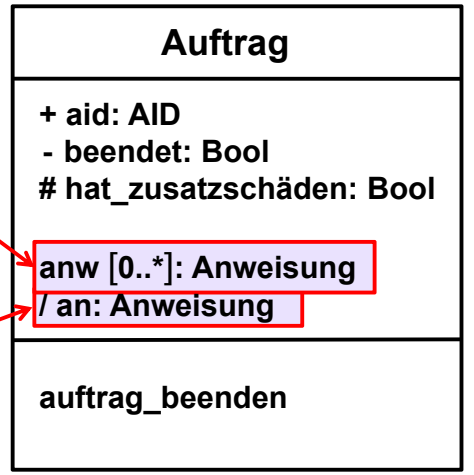
Erweiterte Syntax für Attribute
Multiplizität und abgeleitete Attribute

Zeichen	Bedeutung	Umschreibung
[2..*],[*], ...	multiplicity	Definiert für mehrwertige Attribute ein Intervall, wie viele Werte aufgenommen werden können.
/	derived	Abgeleitete Attribute sind Attribute, deren Ausprägungen aus anderen Attributen berechnet werden können

Erweiterte Syntax für Attribute
Beispiel Multiplizität und abgeleitete Attribute

mehrwertiges Attribut mit Multiplizität (beliebig viele Anweisungen)

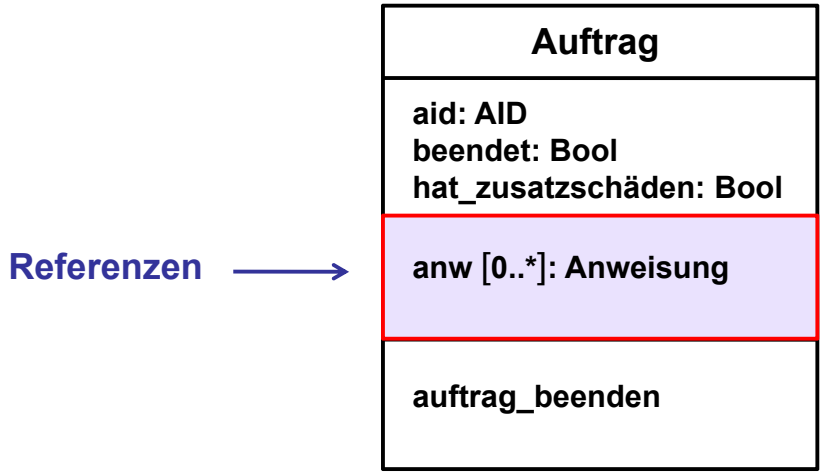
ableitbares Attribut (*an* kann aus *anw* berechnet werden)



Erweiterte Syntax für Attribute
Eigenschaften (attribute-properties)

Eigenschaft	Umschreibung
readOnly	Nach einer Initialisierung kann das Attribut nicht mehr neu beschrieben werden (konstant).
unique	Die Elemente eines mehrwertigen Attributes kommen nur einmal vor.
ordered	Die Elemente eines mehrwertigen Attributes sind geordnet.
sequence/bag	Es gilt die übliche Semantik. Für unique = false und ordered = true gilt sequence implizit.

Wie kommt man systematisch zu Referenzierungsattributen?



Aufgabe von Referenzierungsattributen

- Festlegung der **Rolle** eines Objektes innerhalb des Gesamtsystems
- Information kommt aus den **Kommunikationsdiagrammen**
  1. **Klient-Rolle**
     - Klient-Objekt stellt Anfragen an ein anderes Objekt (Server)
  2. **Server-Rolle**
     - Server nimmt Anfragen von einem anderen Objekt (Klient) entgegen

Klient-Objekte müssen ihre Server-Objekte kennen!
Nur so können Anfragen gestellt werden!

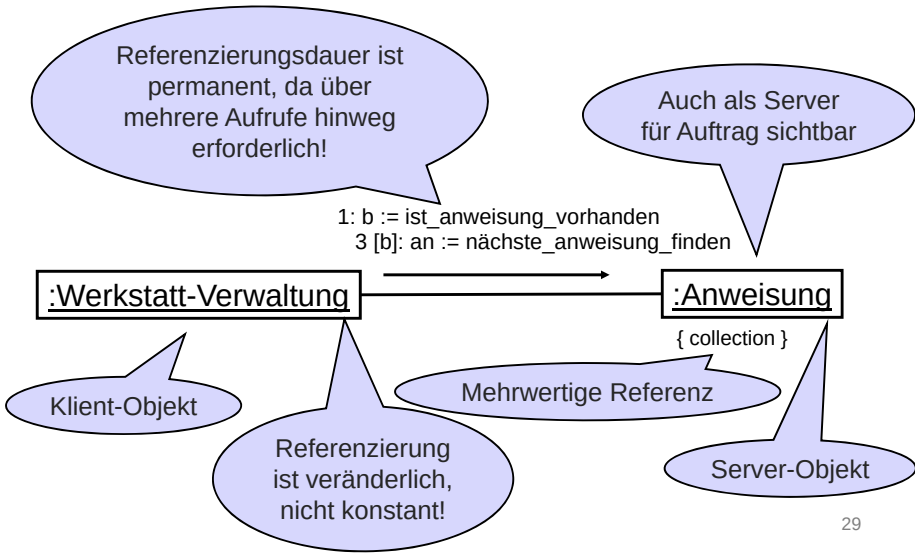
Herleitung von Referenzierungsattributen

1. Identifizieren von Klient-Klassen
- Überprüfen der Kommunikationsdiagramme
- Finden aller Server-Klassen zu einem Klient

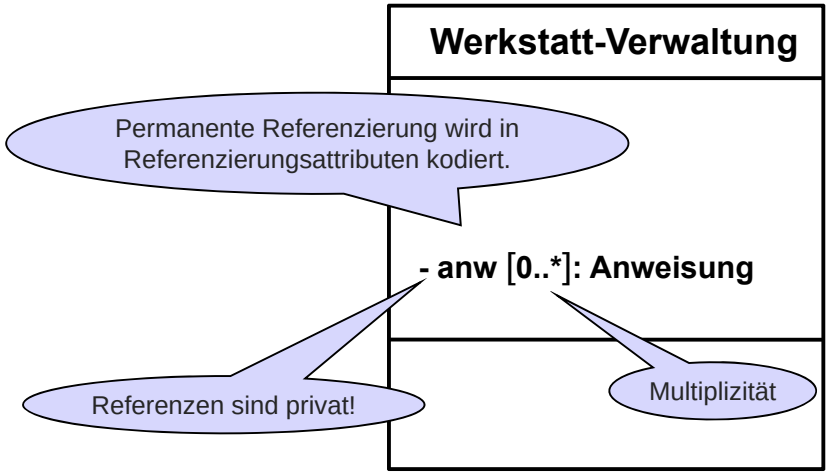
2. Untersuchen der Referenz-Beziehungen
- **Referenzierungsdauer** durch den Klienten
- **Sichtbarkeit** des Servers für seine Klienten
- **Änderbarkeit** der Referenzierung auf den Server
- **Multiplizität** des Servers aus Sicht des Klienten

3. Einführen von Referenzierungsattributen
- Klient-Klassen erhalten Attribute mit Referenzen auf Server, annotiert mit den gewonnen Informationen

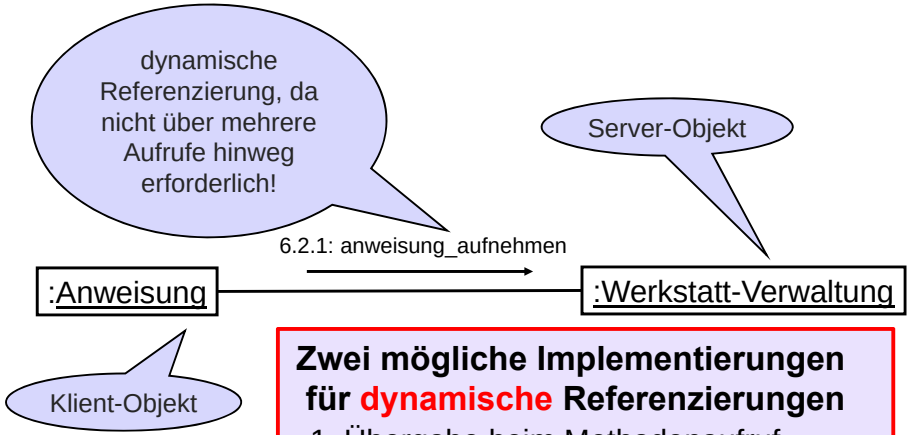
Herleitung von Referenzierungsattributen
Beispiel : Beziehung Werkstatt-Verwaltung und Anweisung



Definition von Referenzierungsattributen
Beispiel: Beziehung Werkstatt-Verwaltung und Anweisung



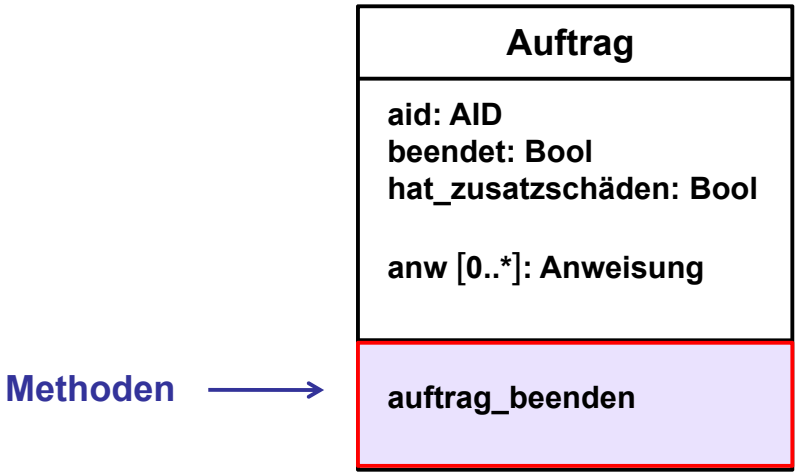
Herleitung von Referenzierungsattributen
Beispiel:Beziehung Anweisung und Werkstatt-Verwaltung



Zwei mögliche Implementierungen für dynamische Referenzierungen

1. Übergabe beim Methodenaufruf
2. Klient kennt jemand, der den Server kennt (speichern in lokaler Variable)

Wie kommt man systematisch zu den erforderlichen Methoden?



Herleitung von erforderlichen Methoden

1. Implementierung von Systemoperationen

- Prüfen, welche Control-Klassen mit der Boundary-Klasse eines Akteurs verbunden sind (siehe Systemklassenmodells)
- Einführen von **Methoden für Systemoperationen in dafür zuständigen Control-Klassen** (siehe Kommunikationsdiagramme)

2. Auffinden systeminterner Methoden

- Prüfen aller Nachrichtenflüsse im Kommunikations-diagramm
- Einführen von **Methoden in Server-Klassen zur Bearbeitung von** systeminternen **Anfragen**

Implementierung von Systemoperationen

Beispiel : Methoden für Werkstatt-Verwaltung

Angabe aller Systemoperationen des Mechanikers, da für ihn nur diese Control-Klasse zuständig ist.

Werkstatt-Verwaltung
- anw [0..*] : Anweisung
+ nächste_Anweisung_holen + reparatur_beendet + zusätzliche_schäden + material_fehlt + auftrag_unzureichend + anweisung_bearbeitet

Hinzufügen systeminterner Methoden

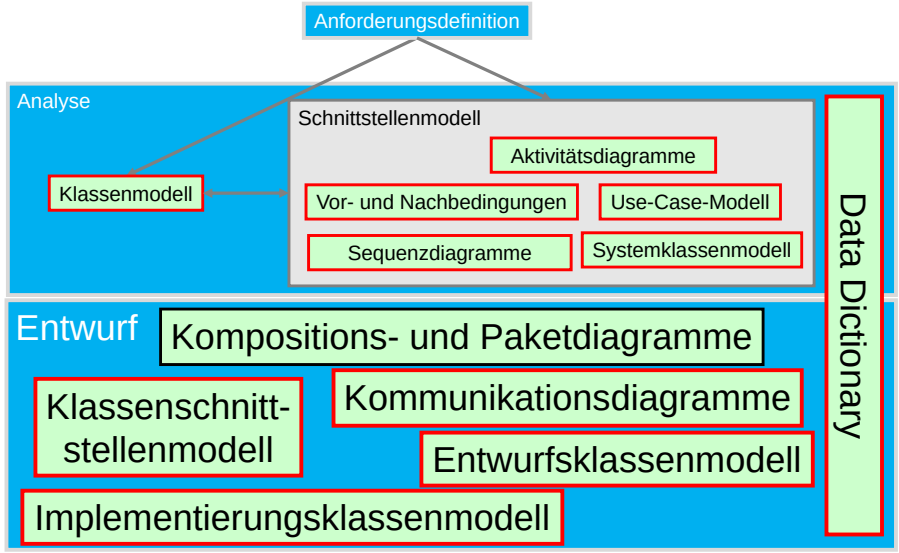
Beispiel : Methoden für Werkstatt-Verwaltung

Angabe aller Systemoperationen des Mechanikers, da für ihn nur diese Control-Klasse zuständig ist.

Anmeldung eines Objekts der Klasse Anweisung an ein Objekt der Klasse Werkstatt-Verwaltung

Werkstatt-Verwaltung
- anw [0..*] : Anweisung
+ nächste_Anweisung_holen + reparatur_beendet + zusätzliche_schäden + material_fehlt + auftrag_unzureichend + anweisung_bearbeitet + anweisung_aufnehmen

Was haben wir bis jetzt erreicht?



Implementierungsklassenmodelle sind Grundlage zur Generierung von Klassenschnittstellen!

Was gehört noch zu einem systematischen Detailentwurf?

Entwurfsmuster

Entwurfsmuster (Design Pattern)

Definition:

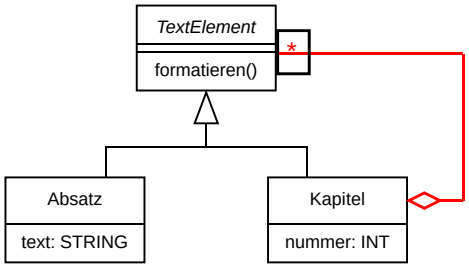
Ein Entwurfsmuster beschreibt eine **häufig auftretende Struktur** von miteinander kommunizierenden Komponenten, die ein **allgemeines Entwurfsproblem in** einem **speziellen Kontext lösen.**

Definition nach Buschmann et al: *Pattern-orientierte Software Architektur,* Addison-Wesley-Longman, 1998.

Rückblick: Aufgaben im Detailentwurf

- **Verfeinerung des ersten Entwurfs**
  - Implementierbarkeit sicherstellen
- **Datenstrukturen festlegen**
  - Realisierung von „Collections“
  - Auswahl der passenden Bibliotheksklasse (List, Map, Set ...)
- **Was ist mit Baumstrukturen?**
  - z.B.: Kapitel, Unterkapitel, Unterunterkapitel ... Absatz

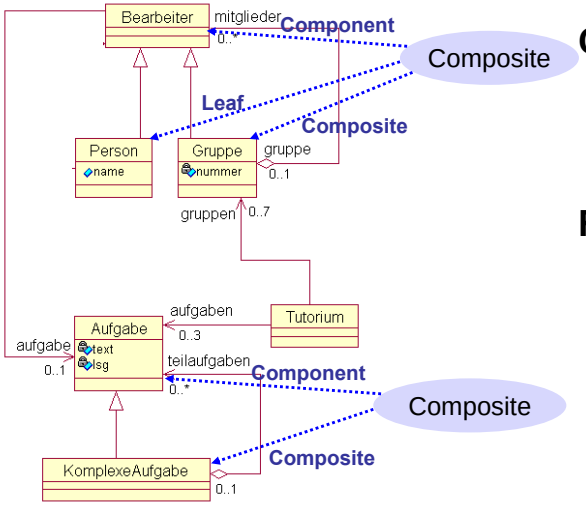
Beispiel für eine Baumstruktur



- Hurra:
  - Aggregation für Rekursion
  - Vererbung für Fallunterscheidung/Rekursionsabbruch

Aber eine Bibliotheksklasse „Baum“ gibt es nicht!

Composite ist ein wiederverwendbares Muster



Composite

- Objekt-Bäume
- Rekursiv
- Einheitlich

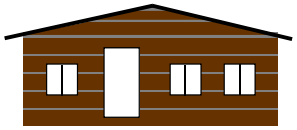
Rollen

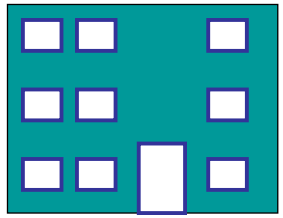
- **Component** (u.U. abstrakt)
- **Leaf** (elementar, konkret)
- **Composite** (zusammengesetzt)

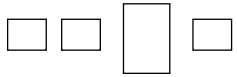
Ursprung von Muster-Wiederverwendung

Vergleiche Architektur

- Fenster, Türen: Baufachhandel
- Aber:





wo kauft man:  ?

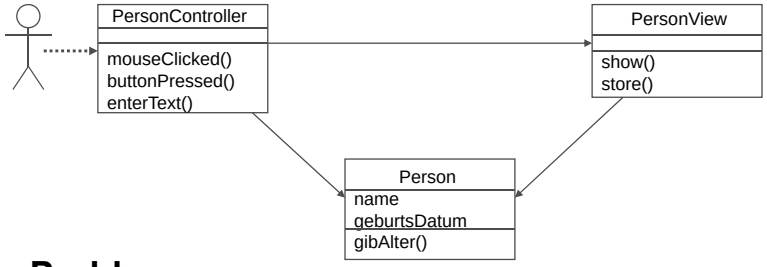
Beschreibung des Musters Composite

Name	Composite
Ziel	Baumstrukturen, einheitliche Behandlung
Motivation	Bsp. Fenstersystem: hierarchische Anordnung v. Elementen bei gleicher Schnittstelle für elementare und komplexe Elemente.
Struktur	(siehe Diagramm)
Rollen	Component, Composite, Leaf (mit Zuständigkeiten)
Kollaboration	Benutzung der Schnittstelle Component, Leaf implementiert direkt, Composite delegiert an Elemente
Anwendungen	Fenstersysteme, Abstrakte Syntaxbäume
Siehe auch	Visitor, Iterator, Decorator, ...
...	

Model-View-Controller (MVC)

- Bewährte Anwendungsstruktur
  Paradigma? Architektur? Entwurfsmuster?
- Im Einsatz seit SmallTalk80
- Klare Trennung
  - Model = Anwendungslogik
  - View = Darstellung (GUI)
  - Controller = Interaktion/Ablaufsteuerung

Model-View-Controller



Problem:

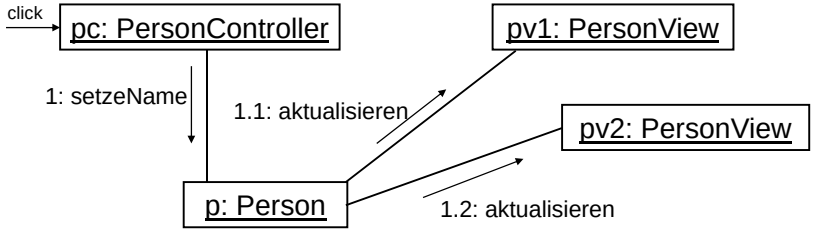
- *Controller* verändert *Model.*
- Wie wird die Änderung in der *View* dargestellt?

Bedingungen:

- *Model* soll *View* nicht kennen!
- *Model* ist u.U. in verschiedenen *Views* dargestellt.

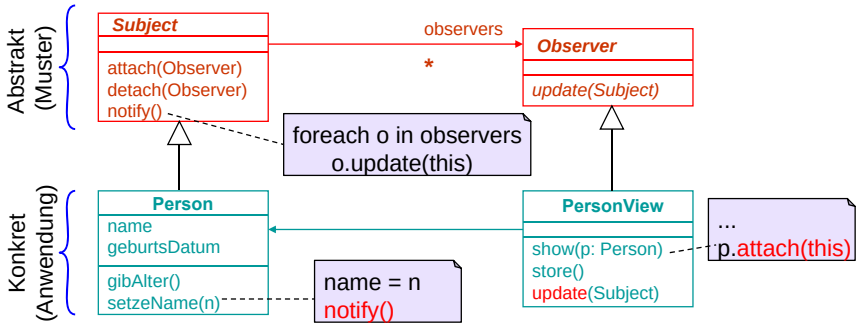
Ablauf: Benachrichtigung

- Abläufe brauchen Objekte und Links
  ⇒ Kommunikationsdiagramm ist geeignet.



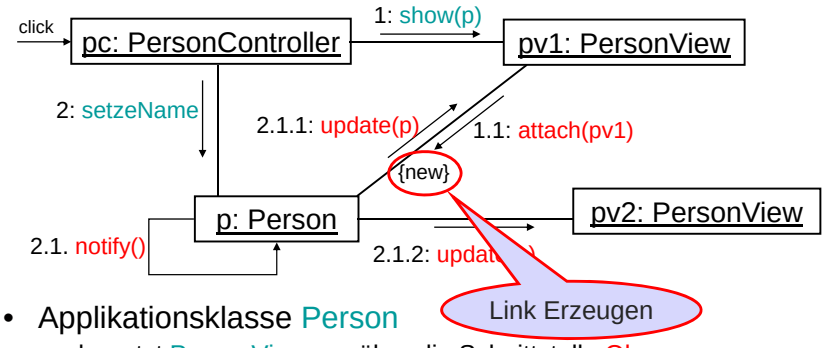
Aber:
wo ist die Unabhängigkeit Person → PersonView?

Muster: Observer



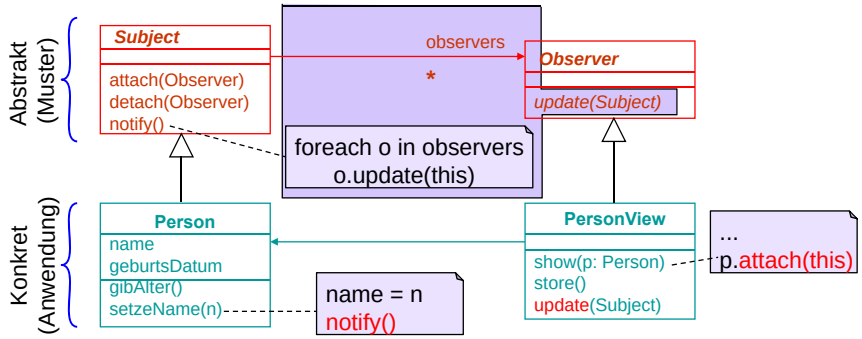
- Durch künstliche Abstraktionen (Subject, Observer) sind Person und PersonView **unabhängiger:**
  - Keine Erwähnung von PersonView in Person.

Observer: vollständiger Ablauf



- Applikationsklasse Person
  - benutzt PersonView nur über die Schnittstelle Observer
  - andere Details sind verborgen: **Abstrakte Kopplung**
- Das Benachrichtigungsprotokoll wird geerbt
  - attach, notify, *(update)*

Muster: Observer



- Durch künstliche Abstraktionen (Subject, Observer) sind Person und PersonView **unabhängiger:**
  - Keine Erwähnung von PersonView in Person.

Observer: Beschreibung

Name	Observer (alias: Listener, Publisher-Subscriber)
Ziel	1:n Beziehung, Benachrichtigung, Unabhängigkeit
Motivation	Bsp. MVC: Konsistenz zw. verschiedenen Sichten, obwohl die Sichten sich nicht (direkt) kennen.
Struktur	(siehe Klassendiagramm)
Rollen	Subject, Observer, ConcreteSubject, ConcreteObserver
Kollaboration	(Siehe **Kommunikationsdiagramm**)
Implement.	Viele Alternativen, z.B. Parameter Art_der_Änderung
Konsequenzen	☺ **abstrakte Kopplung** ☺ Broadcast Kommunikation ☹ unerwartete updates (☹ Verständnis, ☹ Performanz)

Vom Umgang mit Mustern

- **Erfahrungswissen**
  - nicht erfunden, sondern (in guten Lösungen) **ge**funden
- **Schematisch erfassen**
  - einheitliche Beschreibung nach vorgegeben Kategorien
- **Katalogisieren**
  - sammeln, ordnen
  - z.B. Struktur-, Verhaltens-, Erzeugungsmuster
- **Anwenden**
  - die gleiche Lösung stets neu umsetzen: immer ähnlich, niemals exakt gleich!
- **Kommunizieren**
  - Muster = Vokabular

„Gang of Four“

„Katalog“ von Entwurfsmustern

Gamma, Helm, Johnson, Vlissides

„Design Patterns – Elements of Reusable Object- Oriented Software“

➡ Einteilung in …

- Erzeugungsmuster
  - *Abstract Factory*
  - *Builder*
  - *Factory Method*
  - *Prototype*
  - *Singleton*
- Strukturmuster
  - *Adapter*
  - *Bridge*
  - ***Composite***
  - *Decorator*
  - *Facade*
  - *Flyweight*
  - *Proxy*
- Verhaltensmuster
  - *Chain of Responsibility*
  - *Command*
  - *Interpreter*
  - *Iterator*
  - *Mediator*
  - *Memento*
  - ***Observer***
  - *State*
  - *Strategy*
  - *Template Method*
  - *Visitor*