Klassendiagramme und UML

Jens Kosiol

Wintersemester 23/24

(Foliensatz basiert teilweise auf Folien von Prof. Dr. Gabriele Taentzer)

Überblick

- Einführung von Klassendiagrammen
  - Rolle im Entwicklungsprozess
  - Grundlegende Syntax von Klassendiagrammen
  - Erster Entwurf am Beispiel der Study-Right University
- Die *Universal Modeling Language* (UML) und weitere Konzepte für Klassendiagramme
- Die *Object Constraint Language* (OCL) zur Modellierung weiterer Eigenschaften

Generelles Vorgehen

Stories/Scenarios/Beispiele

- Stammen vom Kunden
- So konkret wie möglich
- Grundlage für Objektdiagramme und für Tests

Objektdiagramme

konto42: Konto
kunde = "A. Muster" stand = 400

- Dienen der Kommunikation mit Kunden und Entwicklern
- Snapshot des Heap zur Programmlaufzeit
- Grundlage für Klassendiagramm

Klassendiagramme

- Dienen der Kommunikation zwischen Entwicklern
- Muster für Datenstruktur
- Grundlage für Codegenerierung

Konto
kunde: String stand: float

Code

```
public class Konto {
    public String kunde;
    public float stand;
}
```

Zentrale Konzepte der objektorientierten Modellierung

- Klasse (auch „Objekttyp“):
  - beschreibt die strukturellen und verhaltensmäßigen Merkmale einer Menge gleichgearteter Objekte. Diese Merkmale werden durch Attribute und Operationen (Methoden) beschrieben.
- „Semantische“ Standardbeziehungen (auf Klassen und Objekten) wie z.B.
  - Instanziierung
  - Generalisierung/Spezialisierung
  - Gruppierung (Aggregation oder Komposition)
  - Assoziation

Rolle von Klassendiagrammen

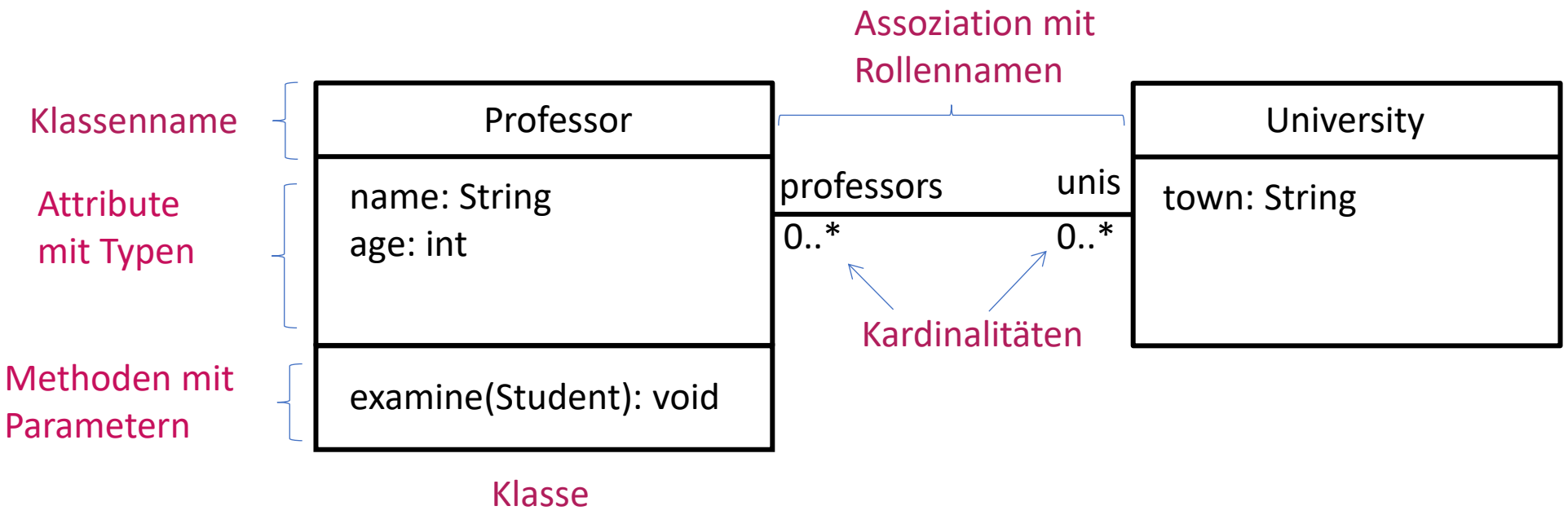
Drei Arten von Modellen

- Analysemodell: Aus Anforderungsbeschreibung/User stories abgeleitet
- Entwurfsmodell: Modell, das vorschreibt, wie ein System zu entwickeln ist
- Implementierungsmodell: Aus existierender Implementierung abgeleitet

Zwecke

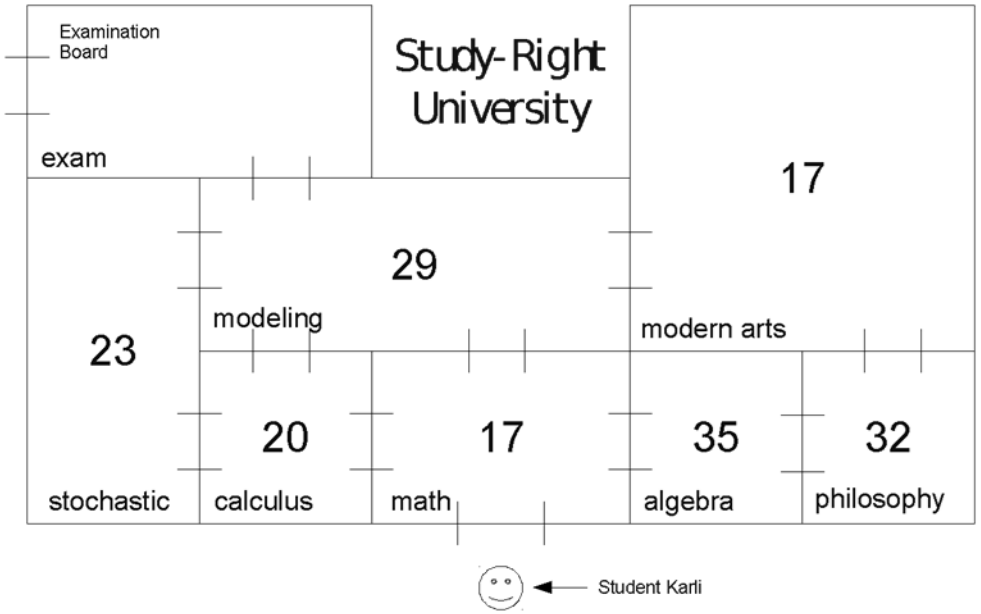
- Dienen der Kommunikation zwischen Entwicklern
- Muster für Datenstruktur
- Grundlage für Codegenerierung
- Graphische Dokumentation
- …

Grundlegende Syntax Klassendiagramm



- Klassennamen beginnen mit **großem Buchstaben**!
- Attributnamen und Bezeichnungen beginnen mit **kleinem Buchstaben**!
- Bei Assoziations-/Rollennamen auf **Einzahl/Mehrzahl** achten!
- Wichtigste Kardinalitäten: **1**, **0..1** oder **0..*** (= *)

Vorlesungsbeispiel: Study-Right University



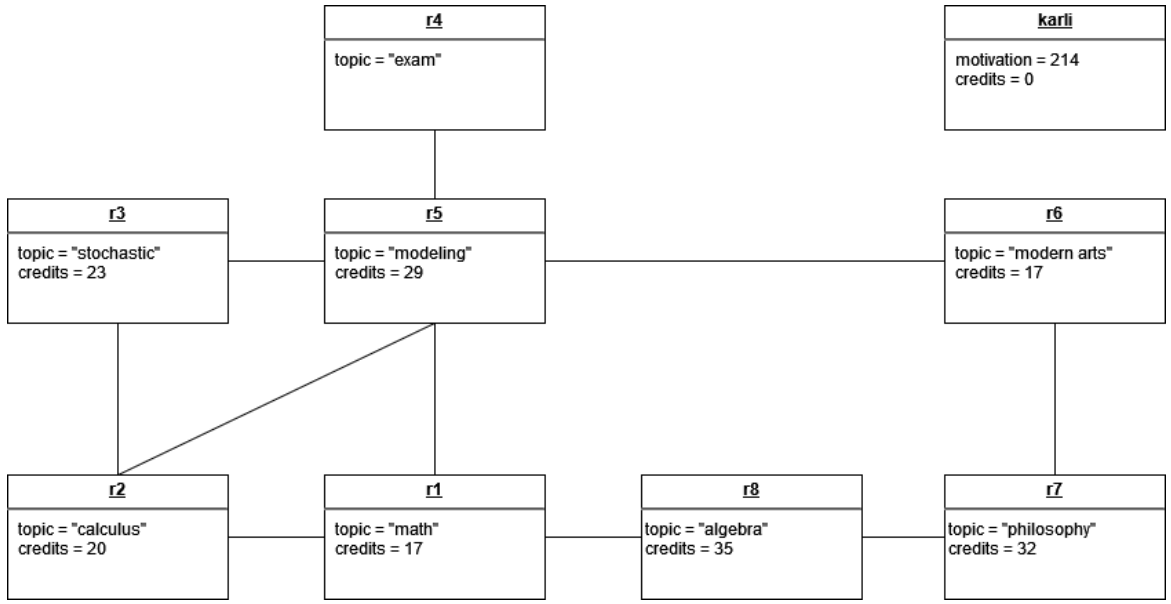
Aufgabe:

- Wegesuche (vom Mathe- zum Examensraum)

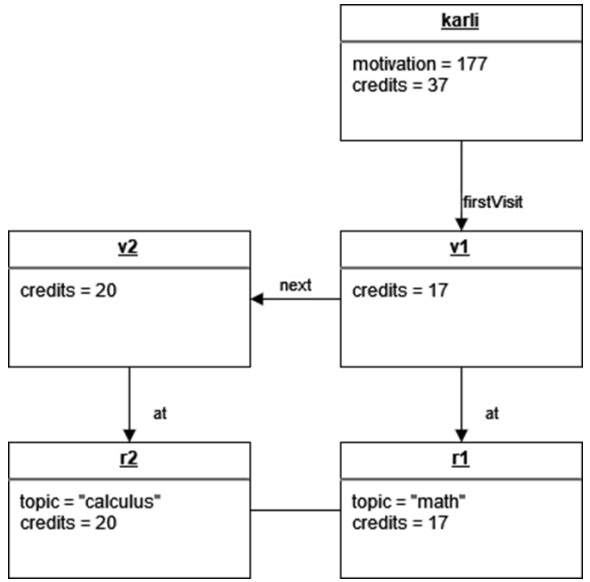
Bedingungen:

- Jeder Credit Point kostet einen Motivationspunkt
- 214 Motivationspunkte gegeben und 214 Credit Points zu erreichen
- Modulabhängigkeiten (Türen zwischen Räumen)
- Mehrfachbelegung möglich (bei Mehrfachbetreten eines Raumes wird jeweils ein anderes Modul unterrichtet)

Welche Klassen brauchen wir?

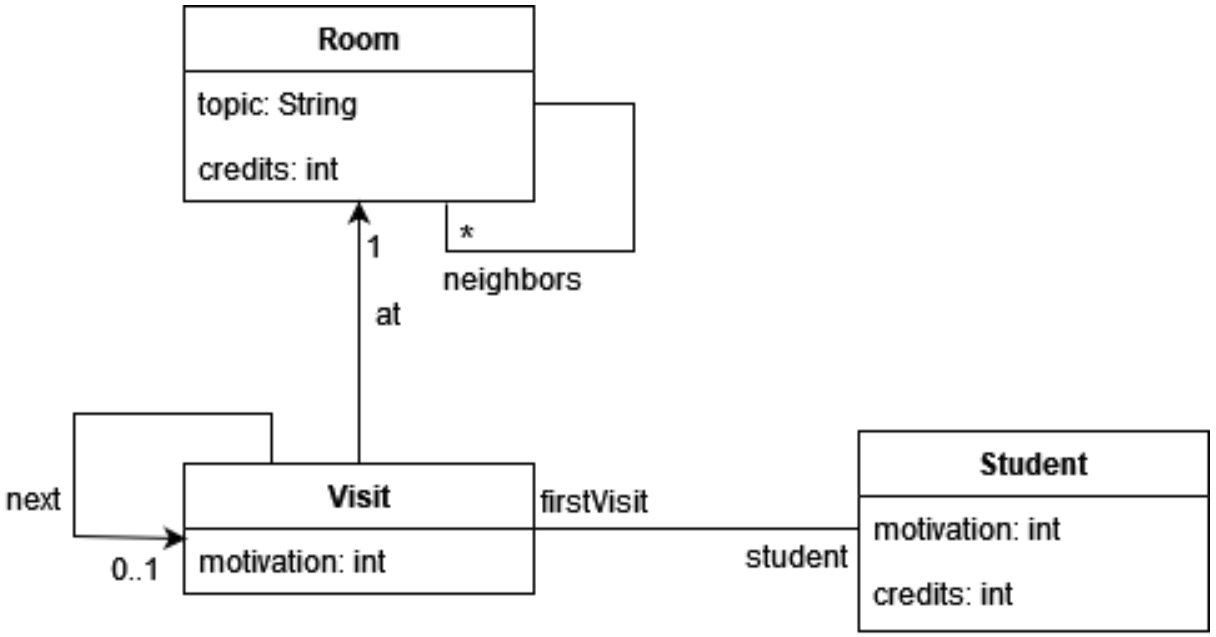


Ausgangssituation im Beispielszenario



Ausschnitt der Objektstruktur nach zwei Schritten von Karli

Einfaches Klassendiagramm für die Study-Right University

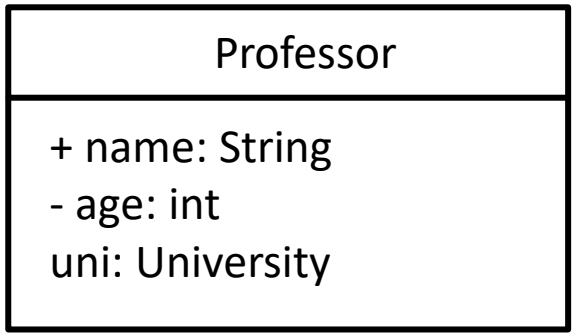


Unified Modeling Language (UML)

- Graphische Modellierungssprache zum Visualisieren des Designs von Systemen
- Entwickelt seit den 90er-Jahren
- Industriestandard [verwaltet von der *Object Management Group* (OMG), ISO genormt]
- Stellt verschiedene Arten von Diagrammen bereit (Klassendiagramme, Objektdiagramme, Komponentendiagramme, Aktivitätendiagramme, Sequenzdiagramme, State Charts, …)
- Aktuelle Version: 2.5.1 (Dezember 2017) https://www.omg.org/spec/UML

Syntax Attribute

Syntax: [*Modifikator*] *Attributbezeichner*: *Attributtyp*



- Der Attributtyp ist nicht weiter spezifiziert. Er kann z.B. ein Klassenbezeichner, ein primitiver Datentyp, ein Typ einer Implementierungssprache oder ein zusammengesetzter Typ sein.
- Attribute können (vor dem Attributbezeichner) bzgl. ihrer Sichtbarkeit gekennzeichnet werden.

Modifikator für Sichtbarkeit:

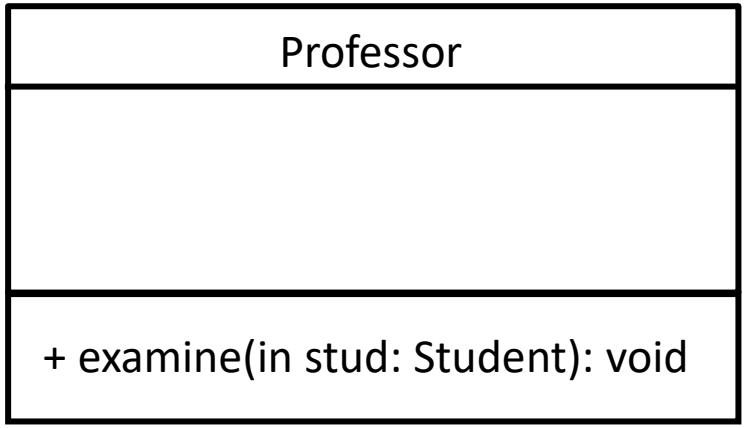
+	öffentlich sichtbar
#	geschützt (in Hierarchie)
~	paketweit
-	nur privat sichtbar

Syntax Operationen

Syntax: [*Modifikator*] *Name*(*Parameterliste*): *Typ*

Parameter:

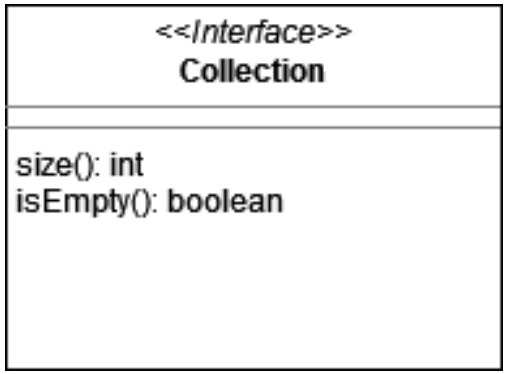
- Syntax: [*Richtung*] [*Name:*] *Typ*
- Mögliche Richtungen: *in* (Default), *out*, *inout*

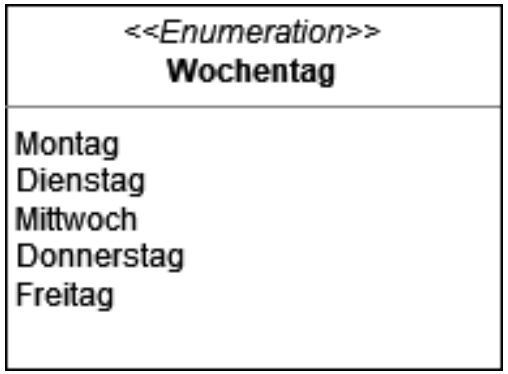


Stereotype

Stereotype spezialisieren ein Modellelement (z.B. eine Klasse) und erweitern das Vokabular von UML.

- Werden in Guillemets (<< … >>) über den Klassennamen gesetzt
- Wichtige Beispiele: <<enumeration>>, <<interface>>
- Dürfen ergänzt werden, um zusätzliche Informationen zu kommunizieren.





Beziehungen zwischen Klassen

In der objektorientierten Modellierung werden verschiedene Arten von Klassenbeziehungen explizit unterschieden. Diese sind:

- *Generalisierung/Spezialisierung* (auch *Vererbung* genannt)

- *Assoziation* (auch *Benutztbeziehung* genannt)

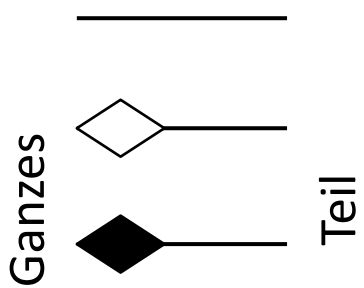
- *gerichtete Assoziation*

- *Abhängigkeit*

Das Ziel einer Generalisierung kann abstrakt sein. Dann wird der Klassenname *kursiv* gedruckt.

Verfeinerung von Assoziationen

Die Stärke einer Assoziation kann durch den Grad einer Ganzes-Teil-Beziehung angegeben werden.



- Assoziation: einfache Verwendung
- Aggregation: Teil trägt zum Zustand des Ganzen bei, kann aber auch unabhängig vom Ganzen existieren
- Komposition: Teil kann nur als Element eines Ganzen existieren

Richtung einer Assoziation

Pfeile am Anfang und Ende einer Assoziation markieren die unterstützte Navigationsrichtung.

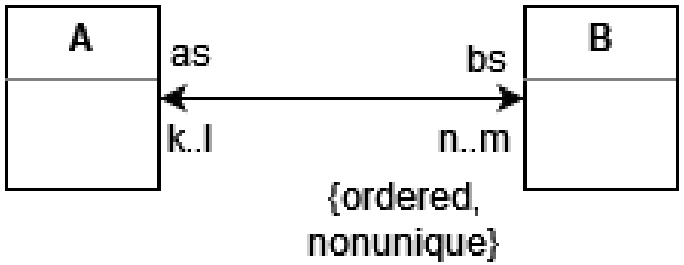
- Navigation nur von Start- zu Zielklasse möglich

- Navigation in beide Richtungen möglich

- Keine Angabe
  (liberale Interpretation: Navigation in beide Richtungen möglich)

Kardinalitäten und Merkmale von Assoziationen

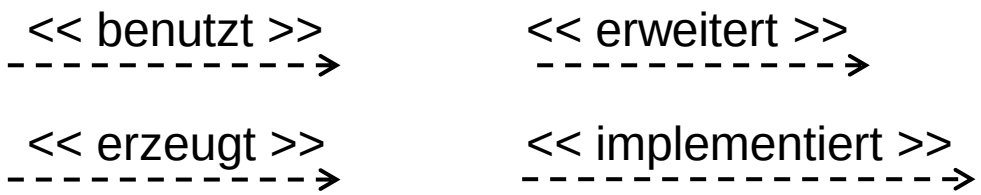
Assoziationen können mit Kardinalitäten und weiteren Merkmalen annotiert werden.



- Sowohl Kardinalitäten als auch Merkmale werden am Ende der Assoziationsrichtung notiert, die sie betreffen!
- Semantik Kardinalität: Von jedem Objekt vom Typ A sollen über Links vom Typ b mindestens n aber höchstens `m` Objekte vom Typ B erreichbar sein.
- Weitere Merkmale können als Kommaseparierte Liste in geschweiften Klammern notiert werden. Wichtige Merkmale: `ordered`, `unique`

Verfeinerung von Abhängigkeiten

Verschiedene Arten von Abhängigkeiten können durch Stereotype kenntlich gemacht werden:



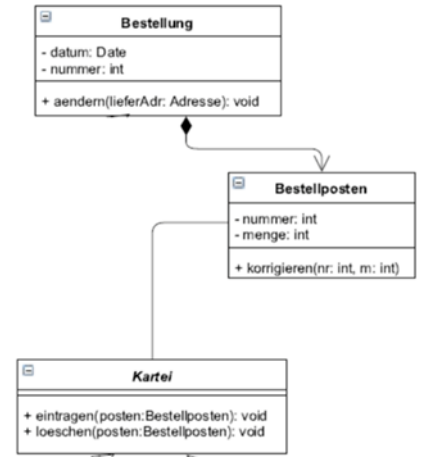
Beziehung zwischen Code und Modellen

- Java-Klasse ↔ Objektklasse
- Java-Feld mit einfachem Datentyp ↔ Attribut in Objektklasse
- Java-Feld mit Objekttyp ↔ Assoziation
- Java-Methode ↔ Operation in Objektklasse
- Java-Vererbung ↔ Generalisierung/Vererbung zwischen Objektklassen

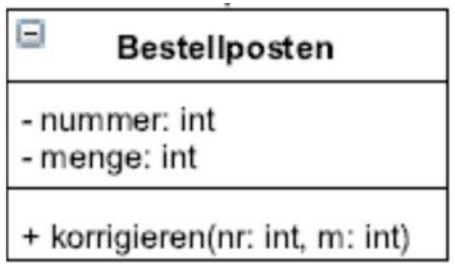
```
public class Beste
   private int num
   private int mer
   public void kor
   //...
   }
}
```

```
public class Bestellung {
   private Date datum;
   private int nummer;
   public List
   //...
   b = new Bes
   //…
}
```

```
public abstract class Kartei {
   //...
   public void eintragen(Bestellposten posten){}
   public void loeschen(Bestellposten posten){}
}
```



Konkrete Klasse



Bestellposten.java

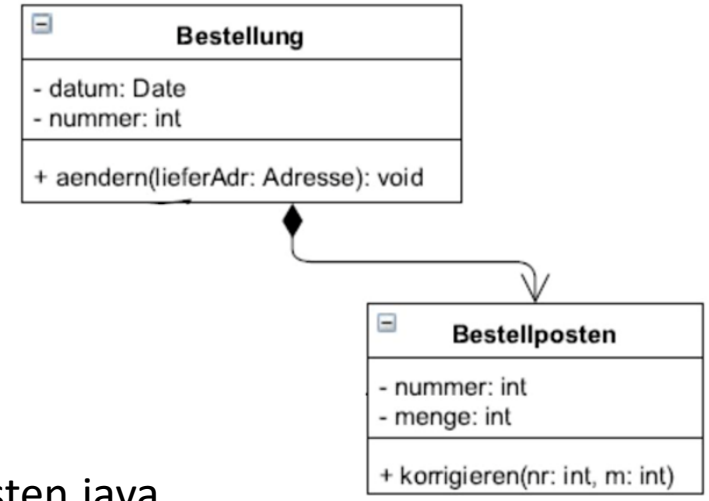
```
public class Bestellposten {
   private int nummer;
   private int menge;
   public void korrigieren(int nr, int m) {
      //...
   }
}
```

Abstrakte Klasse

Kartei.java

```
public abstract class Kartei {
    public void eintragen(Bestellposten posten){
    //...
    }
    public void loeschen(Bestellposten posten) {
    //...
    }
}
```

Komposition



Bestellposten.java

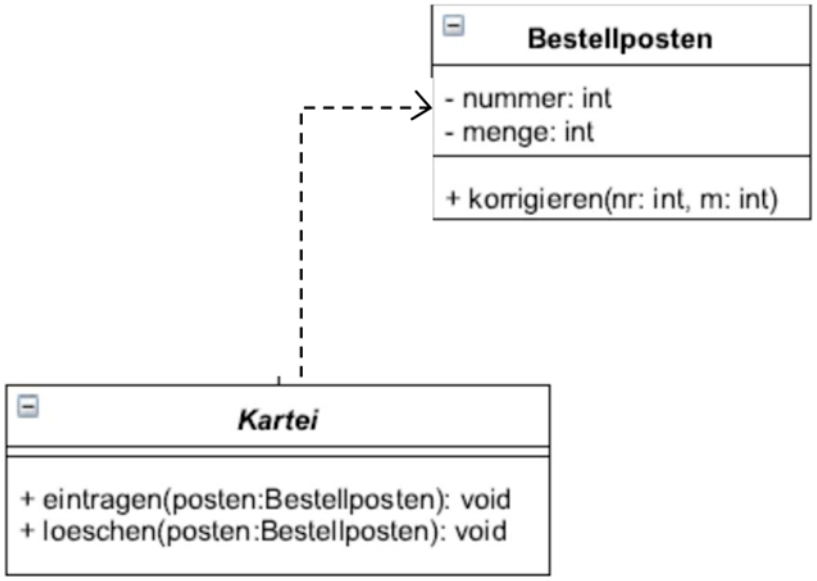
```
public class Bestellposten {
  private int nummer;
  private int menge;
  public void korrigieren(int nr, int m) {
  //...
  }
}
```

Bestellung.java

```
public class Bestellung {
  private Date datum;
  private int nummer;
  public List<Bestellposten> bestellposten;
  // …
  b = new Bestellposten(nummer, menge);
  // …
}
```

Da gerichtete Komposition, kein Feld vom Typ Bestellung in Zielklasse.

Abhängigkeit



Abhängigkeit durch Parameter vom Typ Bestellposten

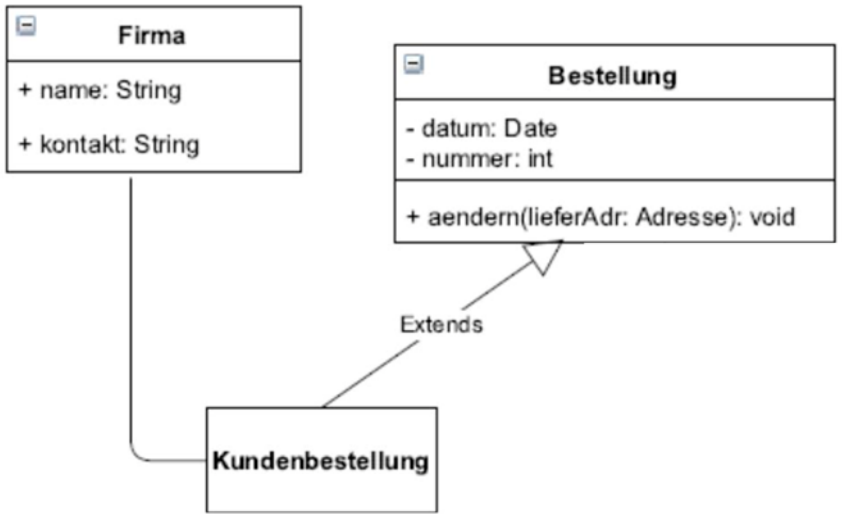
Kartei.java

```java
public abstract class Kartei {
   // …
   public void eintragen(Bestellposten posten){}
   public void loeschen(Bestellposten posten){}
}
```

Bestellposten.java

```java
public class Bestellposten {
   private int nummer;
   private int menge;
   public void korrigieren(int nr, int m) {
   // …
   }
}
```

Vererbung



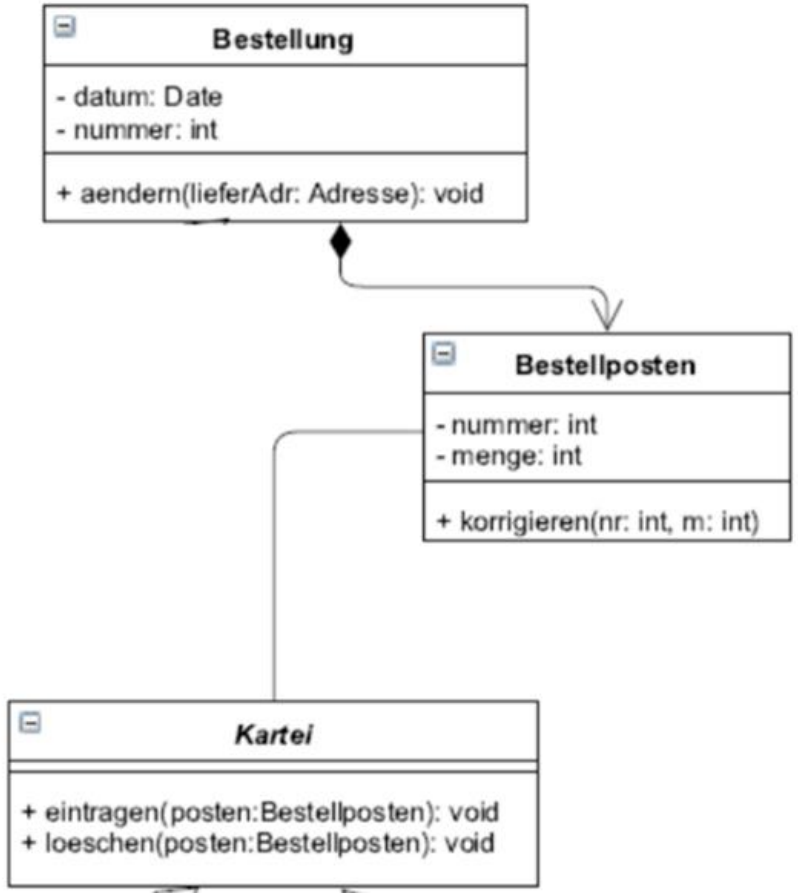
Bestellung.java

```java
public class Bestellung{
    private Date datum;
    private int nummer;
    public void aendern(Adresse lieferadr){
        //...
    }
}
```

Kundenbestellung.java

```java
public class Kundenbestellung extends Bestellung{
    private Firma firma;
    public Kundenbestellung (Firma firma) {
          this.firma = firma;
    }
}
```

Passen Modell und Code zueinander?



Bestellposten.java

```java
public class Bestellposten {
   private int nummer;
   private int menge;
   public Kartei kartei;

   public void korrigieren(int nr, int m) {}
}
```

Kartei.java

```java
public abstract class Kartei {
   //...
   public void eintragen(Bestellposten posten){}
   public void loeschen(Bestellposten posten){}
}
```

Ausdrucksstärke Klassendiagramme

Klassendiagramme geben die Struktur vor, die zugehörige Objektdiagramme haben dürfen. Welche der folgenden Dinge lassen sich durch ein Klassendiagramm ausdrücken?

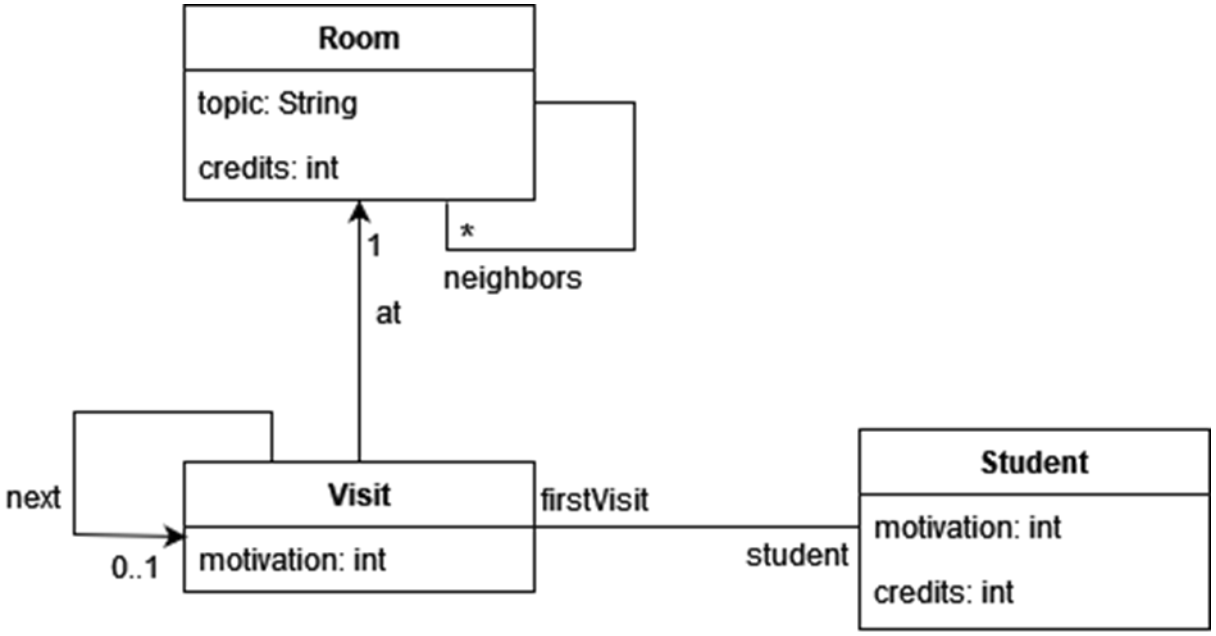
- Nicht-Existenz von Objekten (Beispiel: In keinem gültigen Objektdiagramm gibt es ein Objekt vom Typ A.)
- Existenz von Objekten (Beispiel: In jedem gültigen Objektdiagramm gibt es ein Objekt vom Typ A.)
- Einfache Implikationen zwischen Objekten (Beispiel: In jedem gültigen Objektdiagramm gibt es, wenn es ein Objekt vom Typ A gibt auch ein Objekt vom Typ B.)
- Anforderungen an Attributwerte
- Komplexere Gruppierung von Objekten (Beispiel: Wenn ein Student an Universität A eingeschrieben ist und Kurs B besucht, dann wird Kurs B an Universität A angeboten.)

Die Ausdrucksstärke von Klassendiagrammen reicht alleine oft nicht aus, um die Dinge zu modellieren, die wir modellieren wollen!

Object Constraint Language (OCL)

- Teil des UML Standards
  https://www.omg.org/spec/OCL
- Formale, deklarative Sprache, um Eigenschaften von Objektmodellen zu spezifizieren
  - Invarianten für Objekte
  - Vor- und Nachbedingungen von Operationen
- Basiert auf 4-wertiger Logik höherer Stufe
- (Relativ) einfach zu lesende Syntax

Beispiele OCL



„Ein Student hat nie eine negative Motivation.“

```
Context Student
inv: self.motivation >= 0
```

„Der Mathematikraum wurde (mindestens) einmal besucht.“

```
Context Visit
inv: Visit.allInstances()->
exists(v | v.at.topic =
"math")
```