Programmierung mit Python

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2026-09 Dozent: Karsten Flügge

Themen Heute

- Dataclass
- Objekte Persistieren (speichern csv)
- Verschachtelte Daten & Relation
- Datenbanken (SQL)

Offene Themen / Spezialsyntax

Fehlerbehandlung try
@dekaratoren

lambda (signatur)
Iteratoren + yield
immutable types
list/dict comprehension

@dataclass
@classmethod
@dekarator objekte?

… Vertiefungen

… spezielle module (interne, externe) / Anwendungen

z.B. random, numpy (numerische Berechnungen, Matrizen und Vectoren),

GEMEINSAME INTERSSEN:
GUI, SQL, AI?,

@dataclass

Der dataclass Dekorator ermöglicht es, daten-lastige Klassen **kompakter und typsicher** zu schreiben (ohne __init__) seit **Python 3.7**

```
from dataclasses import dataclass

@dataclass
class Punkt:
  x:int
  y=0 # default

punkt = Punkt(1)
print(punkt.x) # 1
print(punkt.y) # 0
```

@dataclass

Zugriff auf automatisch erzeugte **Attribute**

```
from dataclasses import dataclass

@dataclass
class Punkt:
  x:int
  y=0 # default
```

```
# Felder lassen sich wie gewohnt lesen und schreiben
p.y = 2
print(p.y) # 2

# Neue Felder lassen sich hinzufügen:
p.z = 3
```

```
# convenience: spare __init__ , __str__ , __eq__ … wird für uns automatisch erzeugt
```

immutable dataclass

Veränderungen verhindern mit Argument (**frozen=True**)

```
@dataclass(frozen=True)
class Punkt0:
  x=0
  y=0

p0 = Punkt0() # kann selbst bei Initialisierung nicht gesetzt werden!
p0.x = 1 # AttributeError: can't set attribute
p0.z = 2 # AttributeError: 'Punkt0' object has no attribute 'z'
```

immutable dataclass

Veränderungen verhindern mit Argument (**frozen=True**)

```
@dataclass(frozen=True)
class Punkt0:
  x:int
  y:int

p0 = Punkt0(0,1) # kann bei Initialisierung einmal gesetzt werden!
p0.x = 1 # AttributeError: can't set attribute
p0.z = 2 # AttributeError: 'Punkt0' object has no attribute 'z'
```

Zusatzargumente für Dekorator

repr=True erzeuge Methode zur schönen Darstellung mit print
order=True mache Objekte mit < vergleichbar

complex dataclass

Veränderungen genau steuern mit **field @property (geht auch ohne dataclass!) …**

```
from dataclasses import dataclass, field

@dataclass
class Punkt0:
  _x: int = field(default=0, repr=False)  # Versteckt in der Repräsentation
  y: int = 0

  @property
  def x(self):
    return self._x

  @x.setter
  def x(self, value):
    self._x = value

  def __setattr__(self, name, value):
    if name not in vars(self):
      raise AttributeError(f"{type(self).__name__} hat kein Attribut '{name}'")
    super().__setattr__(name, value)
```

Daten Persistieren II

Bis jetzt haben wir gesehen wie man

- **json** speichert
- text in Datei schreibt

Mit letzterem Ansatz lassen sich auch Objekte wie Person in **csv** Datei schreiben.

Prinzipiell sollte das schon fast jeder hier können, ich zeige es noch mal in IDE

Daten Persistieren II

Bis jetzt haben wir gesehen wie man
- **json** speichert

```
from dataclasses import dataclass, asdict # 3.7

@dataclass
class Punkt:
  x:int
  y:int

  def __repr__(self):
    return str(asdict(self)) # formatierung als dict ≈ JSON

p = Punkt(2,2)
print(p)  # {'x': 2, 'y': 2}

# ACHTUNG, single quotes '' sind NICHT json kompatibel
```

Verschachtelte Daten

Bisher waren unsere Felder (Attribute) stets von einfachen/eingebauten Typen.

Wir können aber natürlich auch unsere eigenen **Klassen(Typen)** als Felder verwenden:

```
class Address:
  def __init__(self, street, city, zip_code):
    self.street = street
    self.city = city
    self.zip_code = zip_code

# Nutze Address in Person:

class Person:
  def __init__(self, name, age, job, address):
    self.name = name
    self.age = age
    self.job = job
    self.address = address

address1 = Address('123 Main St', 'Springfield', '12345')
Person('Alice', 25, 'Engineer', address1)
```

Verschachtelte Daten

Bisher waren unsere Felder (Attribute) stets von einfachen/eingebauten Typen.

Wir können aber natürlich auch unsere eigenen **Klassen(Typen)** als Felder verwenden:

```
class Address:
  def __init__(self, street, city, zip_code):
    self.street = street
    self.city = city
    self.zip_code = zip_code

# Nutze Address in Person:

class Person:
  def __init__(self, name, age, job, address : Address):
    self.name = name
    self.age = age
    self.job = job
    self.address : Address = address

address1 = Address('123 Main St', 'Springfield', '12345')
Person('Alice', 25, 'Engineer', address1)
```

Daten Persistieren III

Verschachtelte Daten lassen sich durch '**ausrollen**' in **csv** Datei schreiben:

```
import csv
with open('people.csv', 'w', newline='') as file:
  writer = csv.writer(file)
  # title row
  writer.writerow(['Name', 'Age', 'Job', 'Street', 'City', 'Zip Code'])
  for person in people:
    writer.writerow([person.name, person.age, person.job, person.address.street, person.address.city,
person.address.zip_code])
```

Das heisst, die Felder von **person.address** werden einfach "**flach**" hinten an geschrieben. Der Kontext, dass diese zur Adresse einer Person gehören ist hier implizit (/verloren).

Man könnte in title row die vollen "Pfade" schreiben
person.name, person.age, person.job, **person.address.street**, person.address.city, person.address.zip_code oder
*# writer.writerow(['Name', 'Age', 'Job', '**Address**.street', 'Address.city', 'Address.zip_code'])*

Für Daten mit mehr Feldern / mehr Verschachtelungen wird es trotzdem bald **unübersichtlich.**

Daten getrennt speichern

Es ist üblich, unterschiedliche Klassen in unterschiedlichen CSV Dateien / Tabellen zu speichern (spätestens wenn Daten zu umfangreich werden, oder wenn SQL verwendet wird)

Dazu muss man aber festhalten welche Daten zusammen gehören.
Hierzu nutzt man eine Objekt ID, auch foreign key / Fremdschlüssel genannt:

```
class Address:

   id_to_address = {} # alle ids zum Auffinden der Adressen

   def __init__(self, street, city, zip_code, id=None):
      self.street = street
      self.city = city
      self.zip_code = zip_code
      if id in id_to_address: raise Exception("duplicate id"+str(id))
      self.id = id or len(Address.id_to_address)+1 # automatisch id erhöhen (Anzahl Adressen)
      Address.id_to_address[self.id] = self # speichere Adresse zur id

class Person:
   def __init__(self, name, age, job, address):
      self.name = name
      self.age = age
      self.job = job
      if isinstance(address, Address):
         self.address = address
      else: # address is an id
         self.address = Address.id_to_address[address]
```

Klassen Variablen

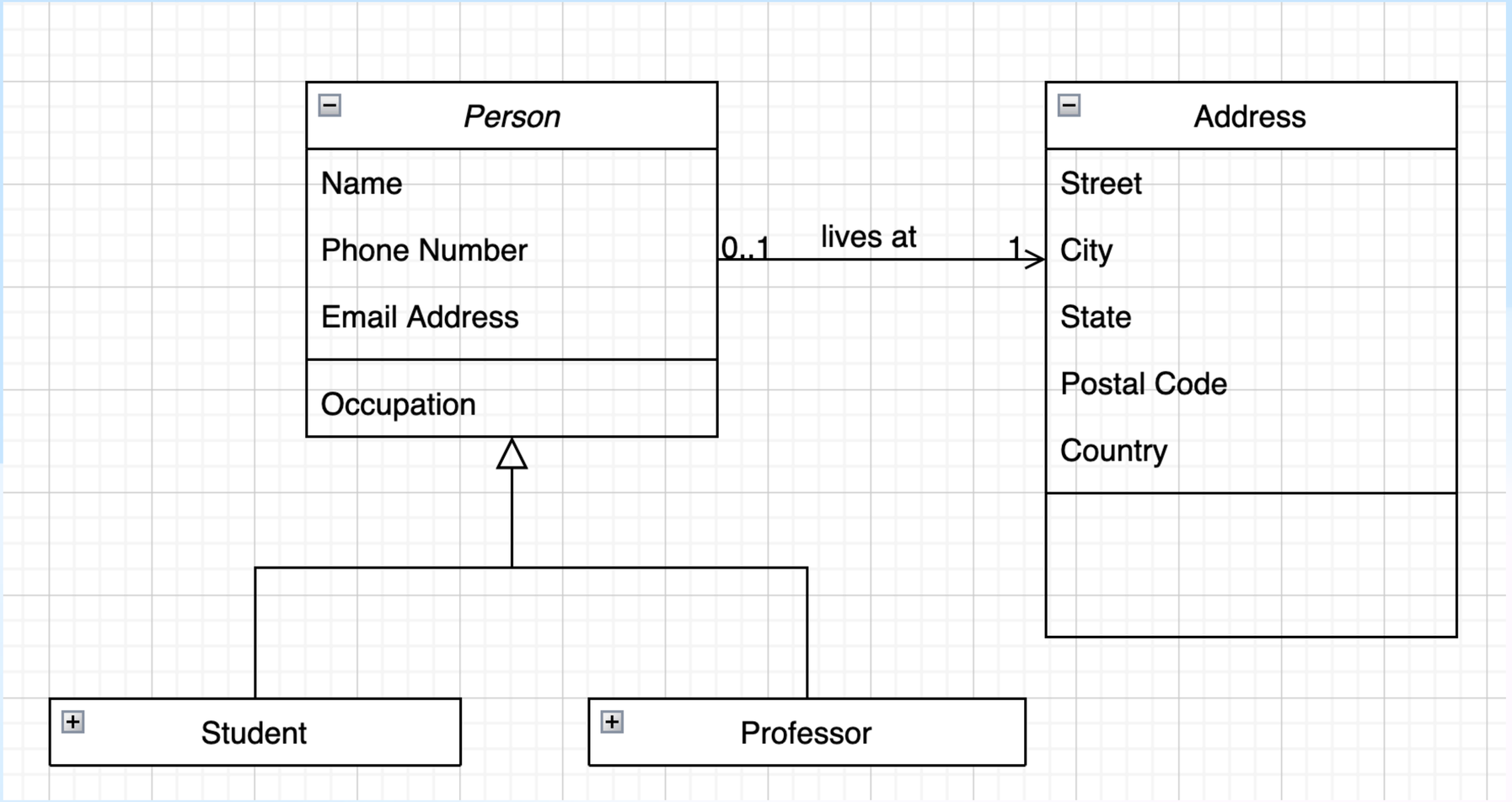
Klassen Variablen stehen oben in class definitionen
Klassen Variablen kann man mit KlassenNamen und Punkt ansprechen,
z.B. Address.id_to_address

```
class Address:

   id_to_address = {} # alle ids zum Auffinden der Adressen

   def __init__(self, street, city, zip_code, id=None):
      self.street = street
      self.city = city
      self.zip_code = zip_code
      if id in id_to_address: raise Exception("duplicate id"+str(id))
      self.id = id or len(Address.id_to_address)+1 # automatisch id erhöhen (Anzahl Adressen)
      Address.id_to_address[self.id] = self # speichere Adresse zur id

class Person:
   def __init__(self, name, age, job, address):
      self.name = name
      self.age = age
      self.job = job
      if isinstance(address, Address):
         self.address = address
      else: # address is an id
         self.address = Address.id_to_address[address]
```

Daten Relationen

Eine Objekt ID oder foreign key kann in verschiedenen Varianten genutzt werden:

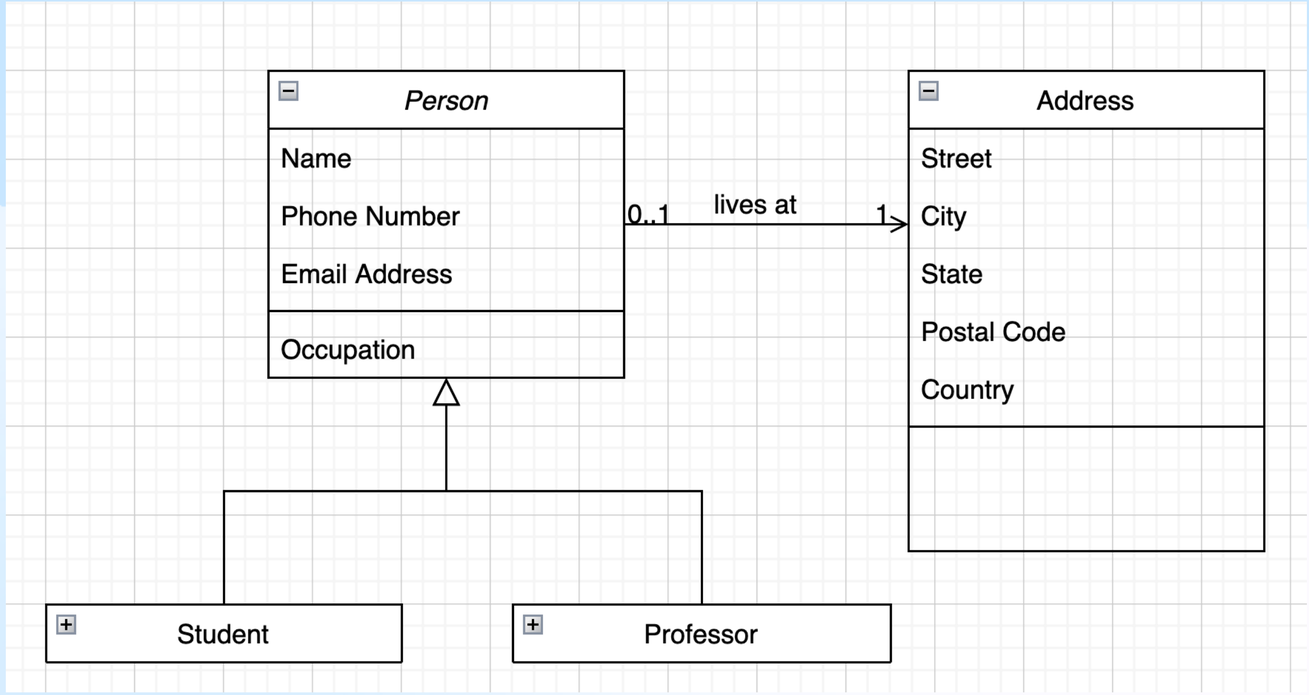
Beispiel: Personen mit Adressen



https://app.diagrams.net/

Klassen Relationen

Relationen: Assoziationen zwischen Typen
// **1-0** Jede Person habe genau eine Adresse (0..1-1 höchstens eine)
1-1 Jede Person habe genau eine Adresse und anders herum
1-n Personen können mehrere Adressen haben
n-1 an einer Adressen können mehrere Personen leben
n-n mehrfach Assoziation: Personen können mehrere Adressen haben, und an einer Adressen können mehrere Personen leben

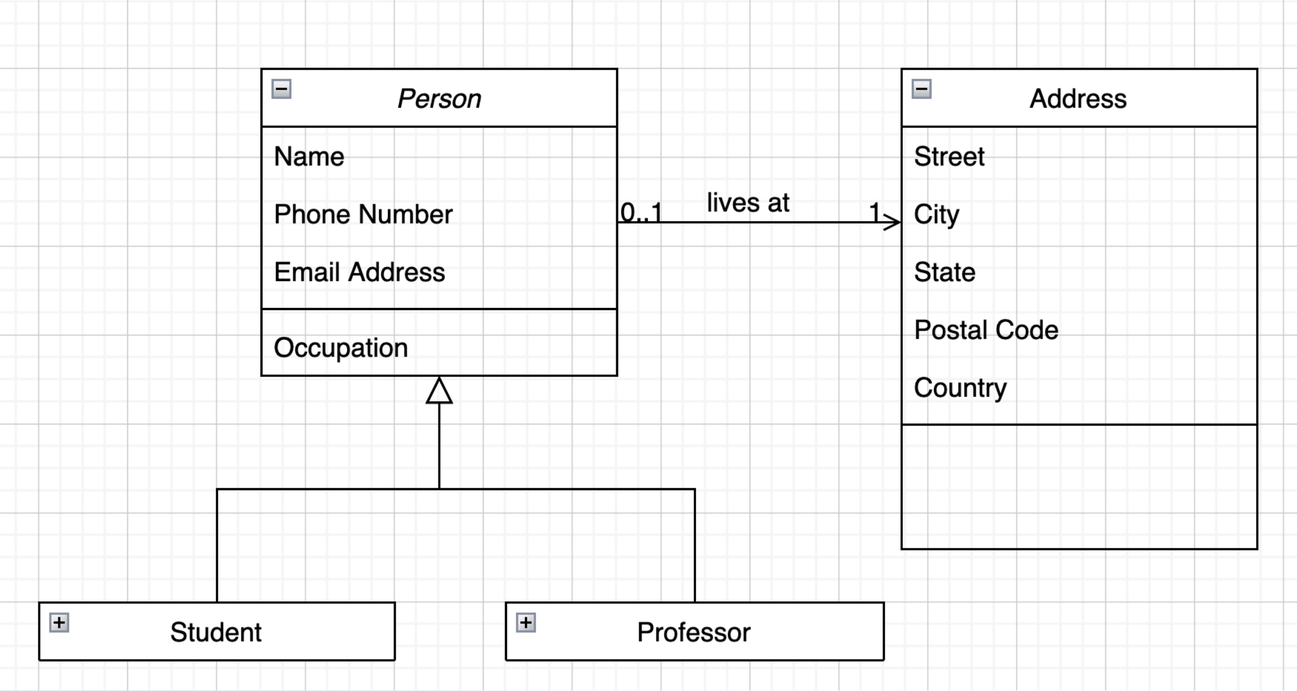


https://app.diagrams.net/

Klassen Relationen

Relationen: Assoziationen zwischen Typen

1-0 "has"
0-1 "belongs to"
1-1 "has and belongs to"
1-n "has many" **author has many books ≈**
n-1 "belongs to many" **author belongs to many books**
n-n "has and belongs to many"



https://app.diagrams.net/

Relationen: Assoziationen zwischen Typen
1-1 Jede Person habe genau eine Adresse
1-n Personen können mehrere Adressen haben
n-1 an einer Adressen können mehrere Personen leben
n-n mehrfach Assoziation: Personen können mehrere Adressen haben, und an einer Adressen können mehrere Personen leben

In Klassen **übersetzt** sich dies in einfache Felder **oder** Listen:

```
@dataclass
class Person:
  address: Address  # 'has' OR
  addresses: list[Address]; # special syntax Type list of Address

@dataclass
class Address:
  person:Person  # 'belongs to one' OR
  persons: list[Person];
```

SQL - eigentlich '**out of scope**' für diesen Kurs, hier trotzdem eine oberflächliche Einführung

SQL steht für "**Structured Query Language**" und ist eine standardisierte Programmier**sprache**, die speziell für das Verwalten und Abfragen von Daten in relationalen Datenbanken entwickelt wurde.

1. Tabellen und Schemata

Tabellen: Daten in SQL-Datenbanken werden in Tabellen gespeichert. Eine Tabelle ist eine Sammlung von Daten, die in Zeilen und Spalten organisiert sind.
Schema: Definiert die Struktur einer Datenbanktabelle und umfasst Definitionen über Datentypen, Spalten und mögliche Beziehungen zu anderen Tabellen.

2. Grundlegende SQL-Befehle

```sql
CREATE TABLE Person (
    Name VARCHAR(100),
    Age INT,
    Job VARCHAR(100),
    AddressID INT,
    FOREIGN KEY (AddressID) REFERENCES Address(AddressID)
);
```

```sql
INSERT INTO Personen (Name, Alter) VALUES ('Anna', 30);
```

```sql
SELECT Name, Alter, Job FROM Personen WHERE Alter > 20;
```

```
# Personen in SQL speichern:

import sqlite3
connection = sqlite3.connect('people.sqlite')
cursor = connection.cursor()

# Save the list of people to the database
for person in people:
  cursor.execute('INSERT INTO Person VALUES (?, ?, ?, ?)',
                 (person.name, person.age, person.job, person.address.id))


besser automatisch:
session.add(User(name="Bob", age=25, address=home))
```

IMPLIZITES SQL

Die SQL Sprache explizit als Text zu benutzen ist Fehleranfällig und unelegant. Besser ist es, unsere Daten per
Data Mapping / ORM (Object-Relational Mapping) Bibliotheken
in SQL-Datenbanken zu speichern ohne selbst SQL zu schreiben!

Beliebte ORM-Bibliothek in Python: **SQLAlchemy**

Kompatible Klassen müssen extra alchemy spezifische Struktur haben:

```
class Address(Base):
    __tablename__ = 'address'
    id = Column(Integer, primary_key=True)
    person_id = Column(Integer, ForeignKey('person.id'))
    location = Column(String)
    person = relationship("Person", back_populates="addresses")
```

Beispiel siehe **save_to_sqlite_alchemy.py**

… out of scope

Data Mapping / ORM (Object-Relational Mapping) Bibliothek
Alternative zu alchemy: django (Teil von Server Framework)

```python
from django.db import models

#django 'dataclass' model
class Person(models.Model):
  name = models.CharField(max_length=100)
  age = models.IntegerField()
  job = models.CharField(max_length=100)


class Address(models.Model):
  person = models.ForeignKey(Person, on_delete=models.CASCADE, related_name='addresses')
  address = models.CharField(max_length=255)
```

Aufgaben

Erstelle Klassen für Person und Adresse mit passenden Feldern
('normal' oder als @dataclass)
Schreibe diese in CSV Datei (erstmal 'flach')
Lese diese aus CSV Datei wieder aus

Modifiziere Klassen mit 1-n Relation. Mit **foreign id**
Schreibe diese in CSV Datei
Lese diese aus CSV Datei wieder aus

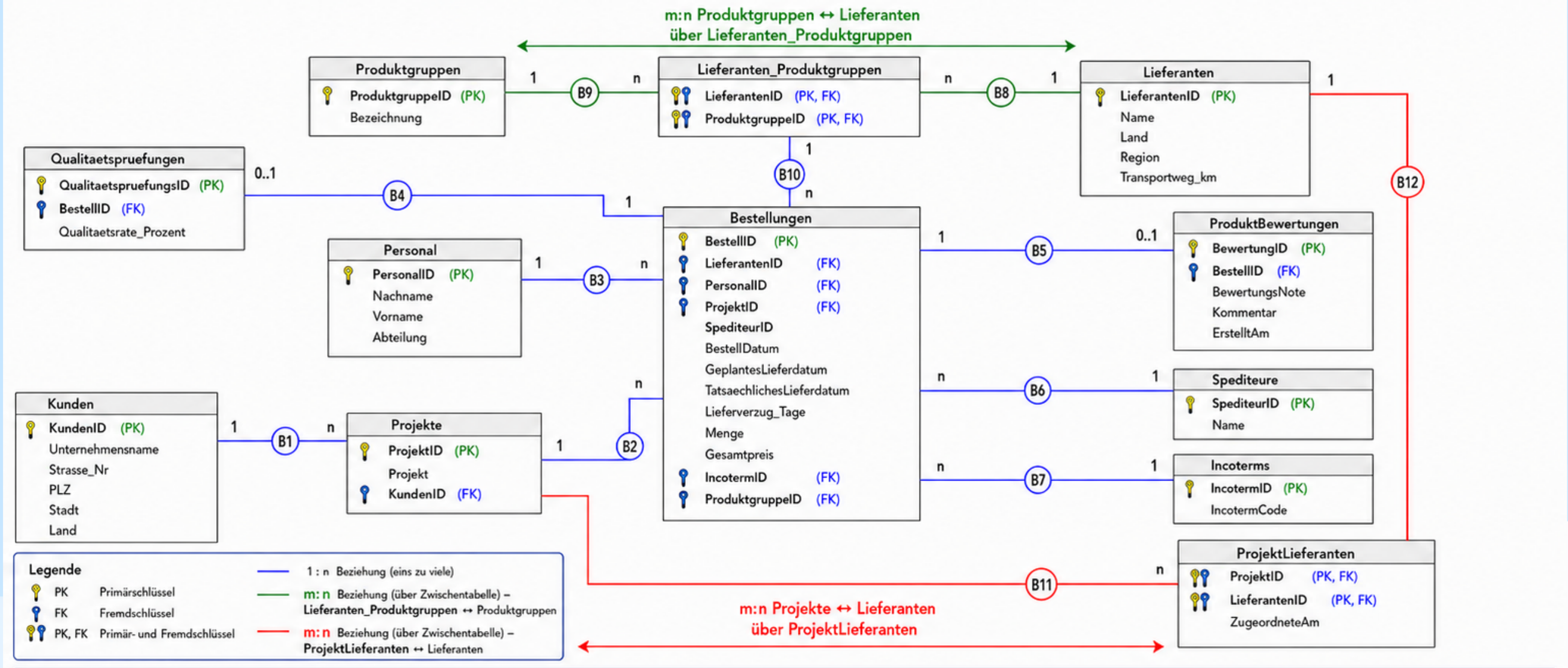
Freiwillig: Speichere obiges in SQL Datenbank und lade wieder.

Zusatzaufgabe

Lustige freiwillige Zusatzaufgabe in Paaren / Gruppen:

Entwerft Klassendiagramm für **Fabrik** mit 2 **Produkten** (mit 3 Attributen). Eine Partei speichert Daten, schafft andere Partei, **auf Anhieb** Daten korrekt zu lesen? Gefragt: genaue **Spezifikation** und **Kommunikation** im Vorfeld.

Beispiel:



Erklärung der Schlüssel im Schema

1. ProduktgruppenID (PK) – Produktgruppen
   Eindeutiger Schlüssel für jede Produktgruppe.
2. LieferantenID (PK) – Lieferanten
   Eindeutiger Schlüssel für jeden Lieferanten.
3. LieferantenID, ProduktgruppenID (PK, FK) – Lieferanten_Produktgruppen
   Zusammengesetzter Schlüssel für die Zuordnung Lieferant – Produktgruppe.
4. QualitaetspruefungsID (PK) – Qualitaetspruefungen
   Eindeutiger Schlüssel für jede Qualitätsprüfung.
5. BestellID (PK) – Bestellungen
   Eindeutiger Schlüssel für jede Bestellung.
6. PersonalID (PK) – Personal
   Eindeutiger Schlüssel für jeden Mitarbeiter.
7. ProjektID (PK) – Projekte
   Eindeutiger Schlüssel für jedes Projekt.
8. KundenID (PK) – Kunden
   Eindeutiger Schlüssel für jeden Kunden.
9. Spediteur-Name (PK) – Spediteure
   Eindeutiger Schlüssel für jeden Spediteur.
10. IncotermID (PK) – Incoterms
    Eindeutiger Schlüssel für jedes Incoterm.
11. BewertungID (PK) – ProduktBewertungen
    Eindeutiger Schlüssel für jede Bewertung.
12. Spediteur – Bestellungen
    Verweist auf den Spediteur über Spediteure.Name.
13. LieferantenID, ProduktgruppeID (FK) – Bestellungen
    Gemeinsam zusammengesetzter Fremdschlüssel (LieferantenID, ProduktgruppeID) auf Lieferanten_Produktgruppen.

Erklärung der Beziehungspfade

- B1 Kunden 1 : n Projekte
- B2 Projekte 1 : n Bestellungen
- B3 Personal 1 : n Bestellungen
- B4 Bestellungen 1 : 0..1 Qualitaetspruefungen
- B5 Bestellungen 1 : 0..1 ProduktBewertungen
- B6 Spediteure 1 : n Bestellungen
- B7 Incoterms 1 : n Bestellungen
- B8 Lieferanten 1 : n Lieferanten_Produktgruppen
- B9 Produktgruppen 1 : n Lieferanten_Produktgruppen
- B10 Lieferanten_Produktgruppen 1 : n Bestellungen – zusammengesetzter Fremdschlüssel (LieferantenID, ProduktgruppeID)
- B11 Projekte 1 : n ProjektLieferanten
- B12 Lieferanten 1 : n ProjektLieferanten