Muster in der Software–Technik

Grundlegende Konzepte der Software–Entwicklung und Objekt–Orientierung

Grundlagen für die weitere Vorlesung:

- Aktivitäten und Prozesse der Software–Entwicklung
- Objektorientierte Modellierung
  - Objekte, Klassen und Merkmale
  - Assoziation, Aggregation, Komposition
  - Spezialisierung und Generalisierung
- Objektorientierte Programmierung
  - Abstrakte Klassen und Schnittstellen
  - Vererbung, Überschreiben, Polymorphismus und Delegation

Aktivitäten der Software–Entwicklung:

- Analysieren – Was ist die Aufgabenstellung?
- Spezifizieren – Welche Funktion soll das System haben?
- Entwerfen – Welche Struktur (Bestandteile, Abläufe) hat das System?
- Implementieren – Wie sieht der Programmcode des Systems aus?
- Testen – Funktioniert das Programm, wie es soll?
- Nutzen – Kann das Programm so genutzt werden, wie es ist?
- Warten – Wie kann das Software–System modifiziert werden, um veränderten Anforderungen gerecht zu werden?

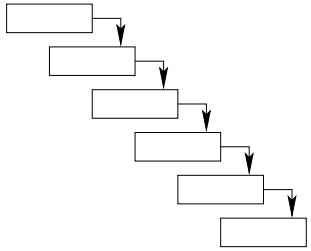
Arten der Software–Entwicklung:

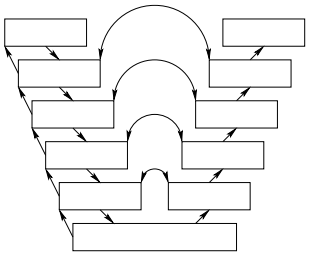
- Neuentwicklung
- Weiterentwicklung/Wiederverwendung

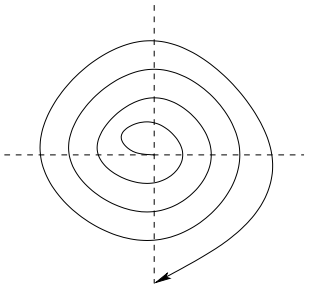
Vorgehensweisen bei der Software–Entwicklung:

- Forward Engineering – aus Modell wird System entwickelt
- Reverse Engineering – aus System wird Modell erzeugt/abstrahiert
- Roundtrip Engineering – Forward und Reverse Engineering wechseln einander ab

Prozessmodelle für die Neuentwicklung von Software:







- Wasserfallmodell: lineares Modell mit phasenweiser Verifikation
- V–Modell: Entwicklung und Validation auf korrespondierenden Abstraktionsebenen
- Spiral–Modell: Zyklische Wiederholung von Planung, Risikoanaylse, Realisierung und Bewertung
- agile Ansätze: Ausmaß an Kontrolle und protokollarischen Vorgängen wird verringert

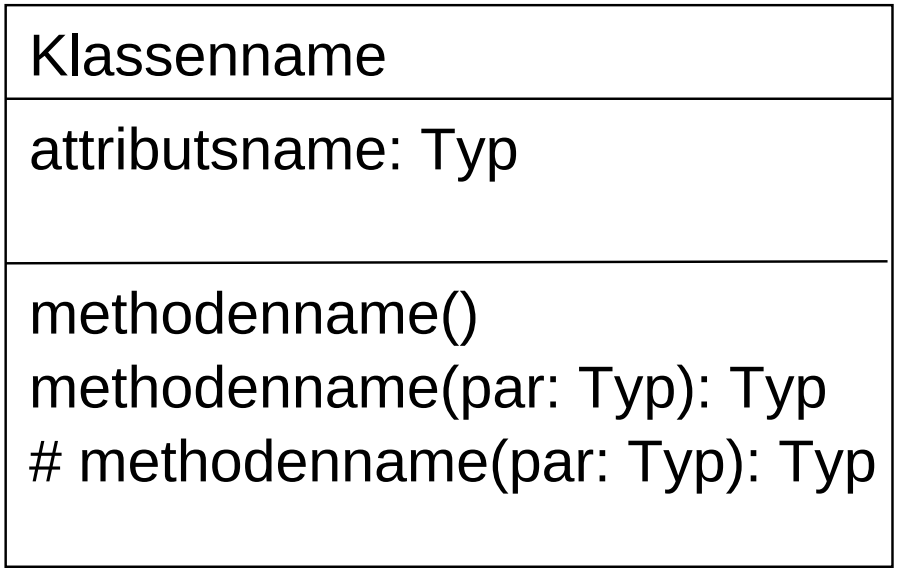
Grundbegriffe der Objektorientierung:

- Objekt
- Merkmale
  - Zustand – Attribute
  - Verhalten – Methoden
  - Lebenszyklus – Konstruktoren
- Beziehungen zwischen Objekten – Assoziationen
- Kategorisierung von Objekten – Klassen

Objektorientierte Modellierung: Werkzeuge in UML

- strukturelle Aspekte
  - Klassendiagramme
  - Objektdiagramme
- dynamische Aspekte
  - Sequenzdiagramme/Interaktionsdiagramme
  - Zustandsdiagramme
  - Aktivitätsdiagramme
- funktionale Aspekte
  - Anwendungsfälle

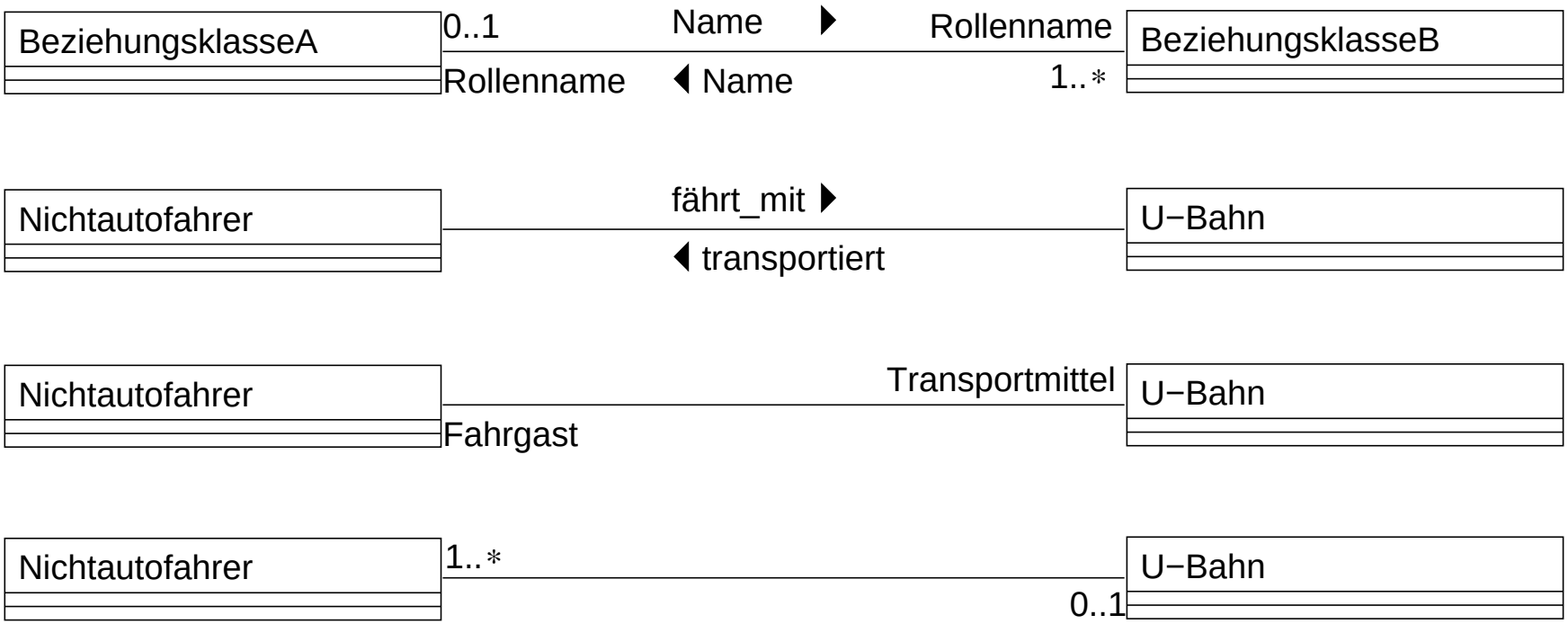
Klassendiagramme in UML: Klassen, Attribute, Methoden



Je nach Modell–Ebene unterschiedlich detaillierte Angaben

Assoziationen:

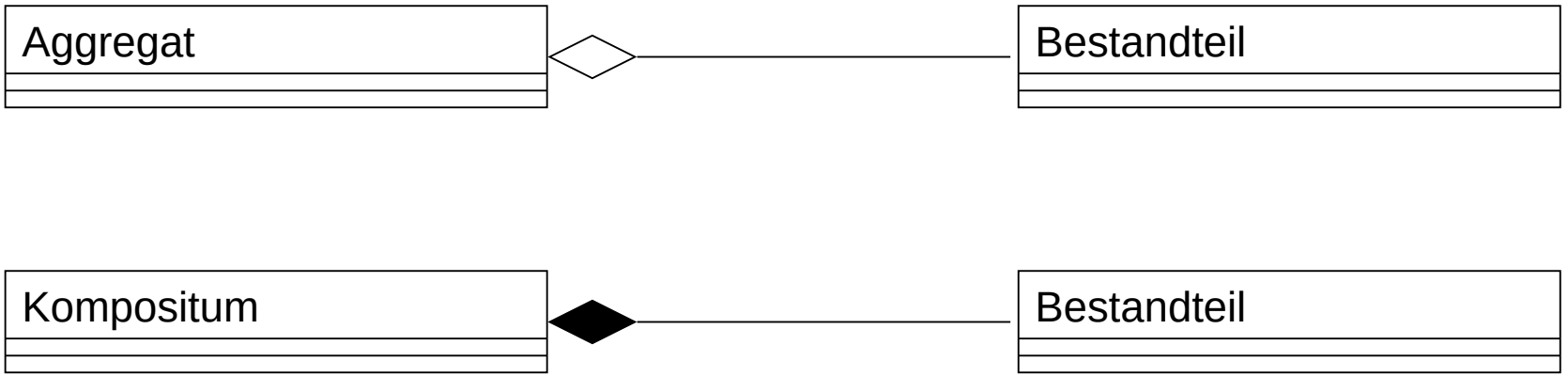
Beziehungen werden durch Assoziationen modelliert:



Die Beziehung besteht zwischen Objekten!

Aggregation und Komposition:

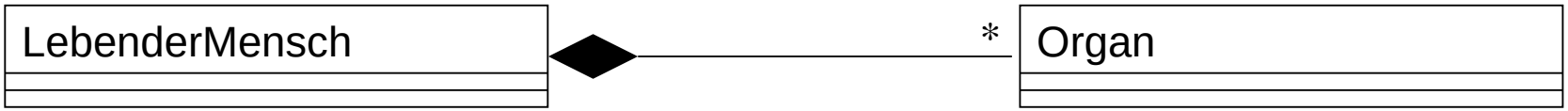
beide Assoziationen drücken die „ist–Bestandteil–von“ bzw. „enthält“–Beziehung aus.



Der Unterschied besteht in der Stärke der Bindung …

Komposition:

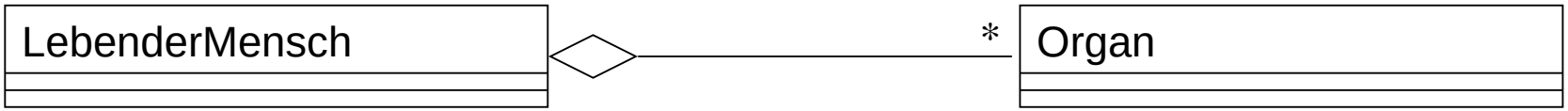
Hier kann der Bestandteil nicht außerhalb des Kompositums existieren:



Medizin vor 100 Jahren

Aggregation:

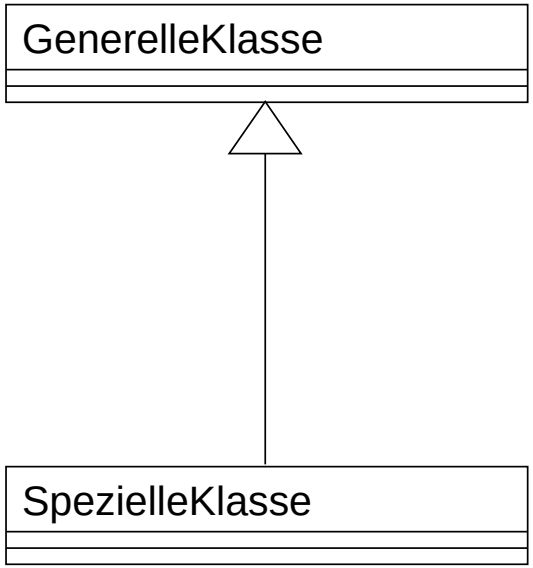
Hier kann der Bestandteil außerhalb des Kompositums existieren:



Medizin heute mit Transplantationen

Bei Aggregation kann das aggregierte Objekt Bestandteil mehrerer Objekte sein (z.B. eine Person in mehreren Vereinen)

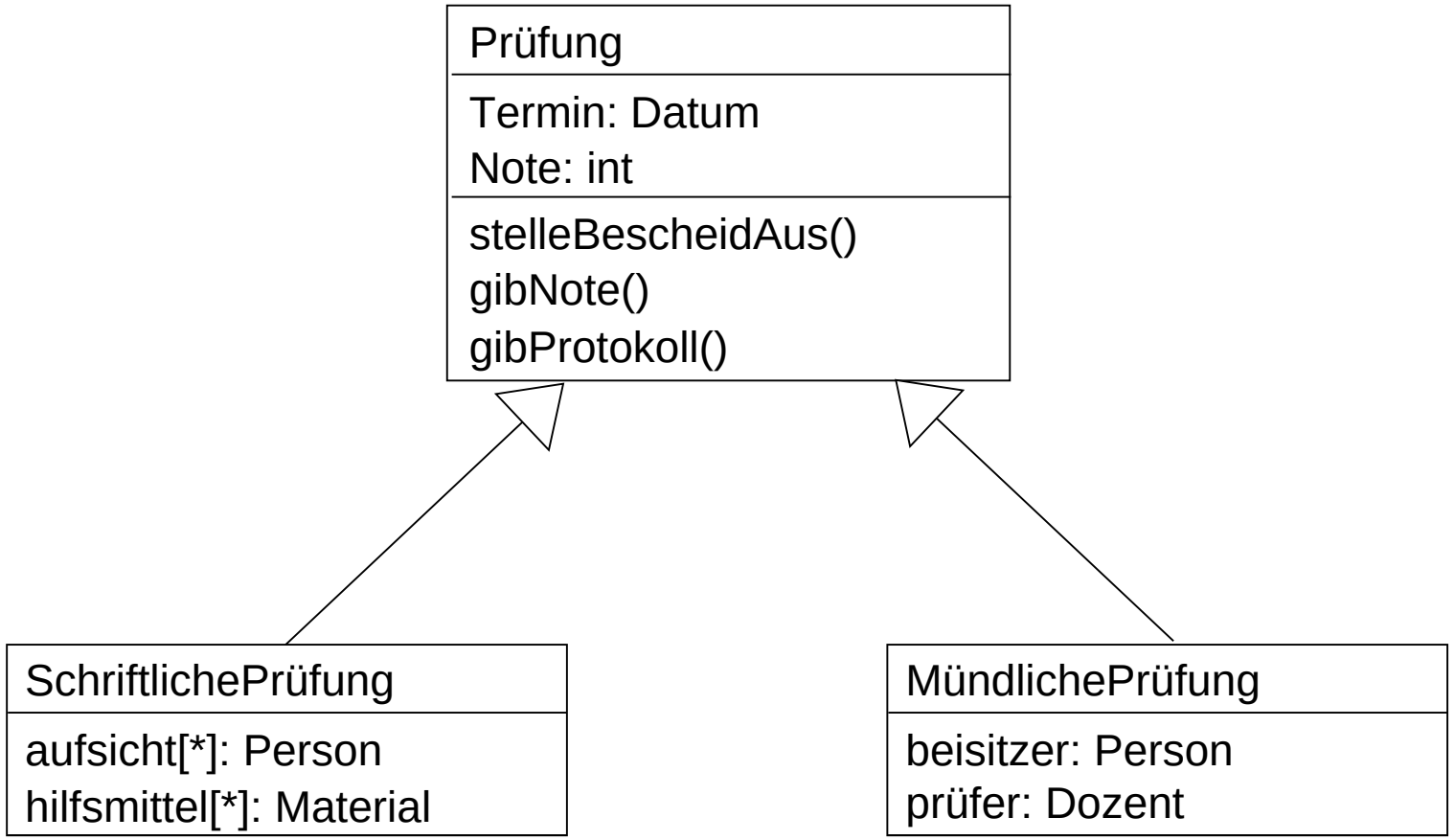
Generalisierung und Spezialisierung:



Die generelle Klasse ist allgemeiner definiert und gibt ihre Merkmale an all ihre speziellen Klassen weiter.

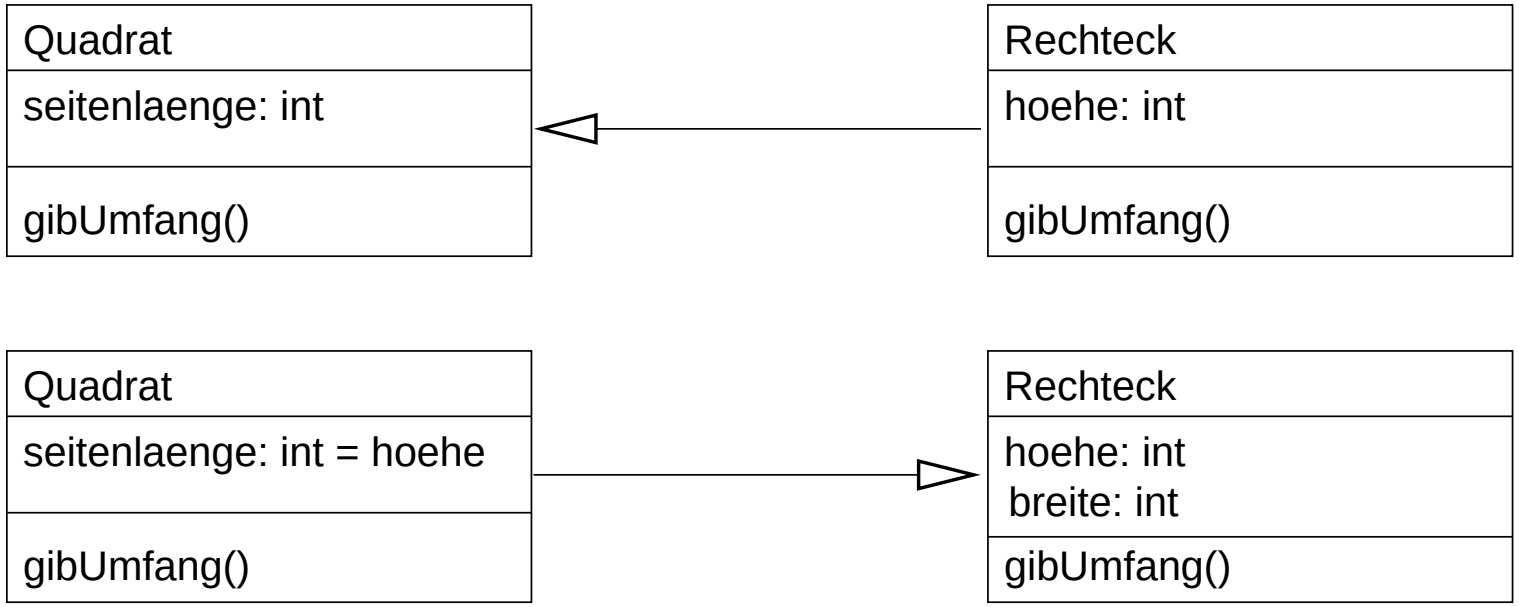
Die spezielle Klasse weist evtl. noch zusätzliche speziellere Merkmale auf

Beispiel zur Generalisierung und Spezialisierung:



Das Henne–Ei–Problem der Objektorientierung:

Rechtecke und Quadrate: Wie modelliert man den Zusammenhang?

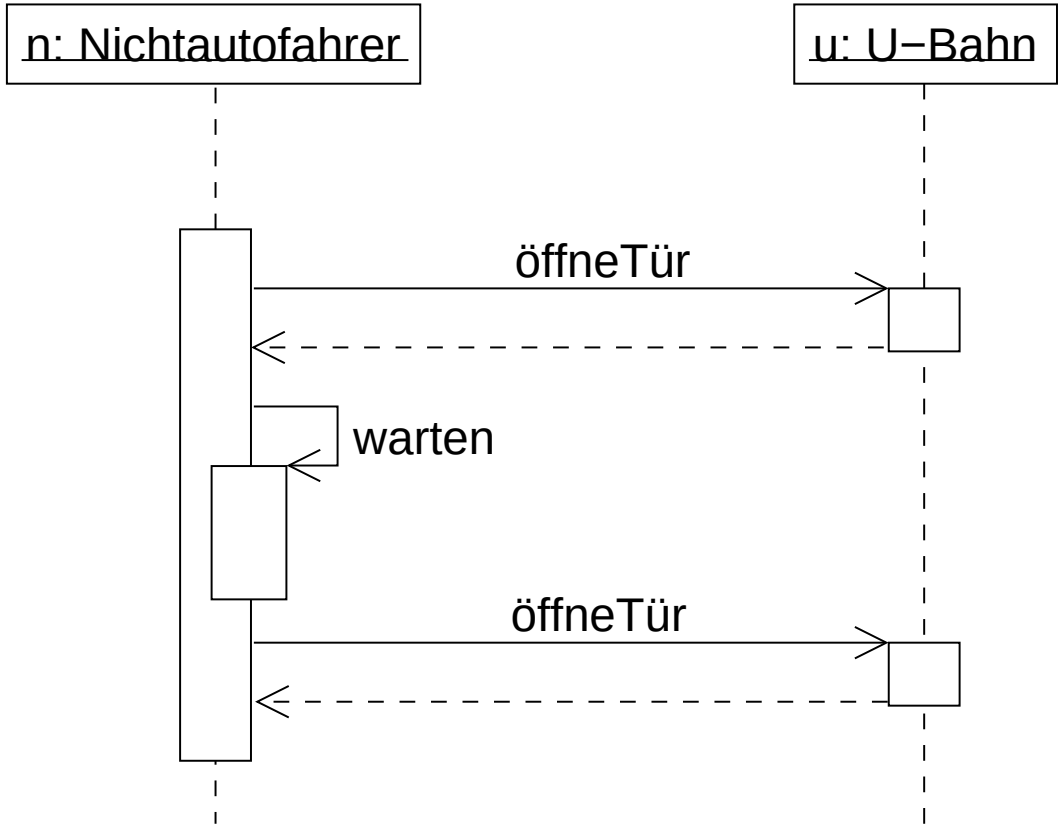


Oder ganz anders?

Modellierung dynamischer Aspekte mit Sequenzdiagrammen:

- Sequenzdiagramme stellen Szenarien für die Interaktion von Objekten dar
- Bestandteile
  - Objekte
    - Lebenslinien, Erzeugung, Zerstörung
    - Aktivierungskästen
  - Nachrichten
- bieten grobe Modellierung von Abläufen, die Vollständigkeit des einzelnen Diagramms (alle Nachrichten eingezeichnet) und der Menge der Diagramme ist nicht gefordert

Modellierung von Interaktionen zwischen Objekten:



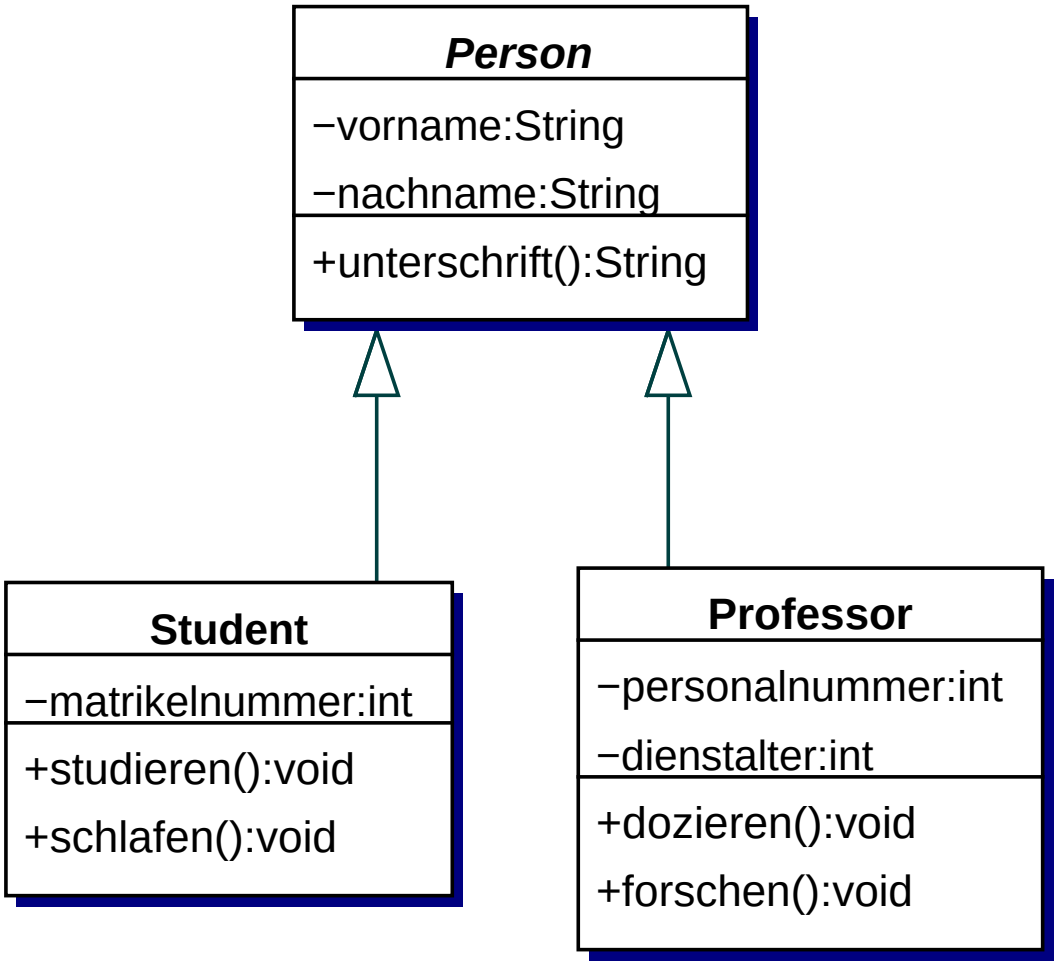
Objektorientierte Programmierung:

- Abstraktion: abstrakte Methoden und Klassen, Schnittstellen
- Vererbung
  - Schnittstellenvererbung
  - Implementierungsvererbung
- Spezialisierung von Methoden
- Polymorphismus
- Wiederverwendung von Eigenschaften: Vererbung vs. Delegation

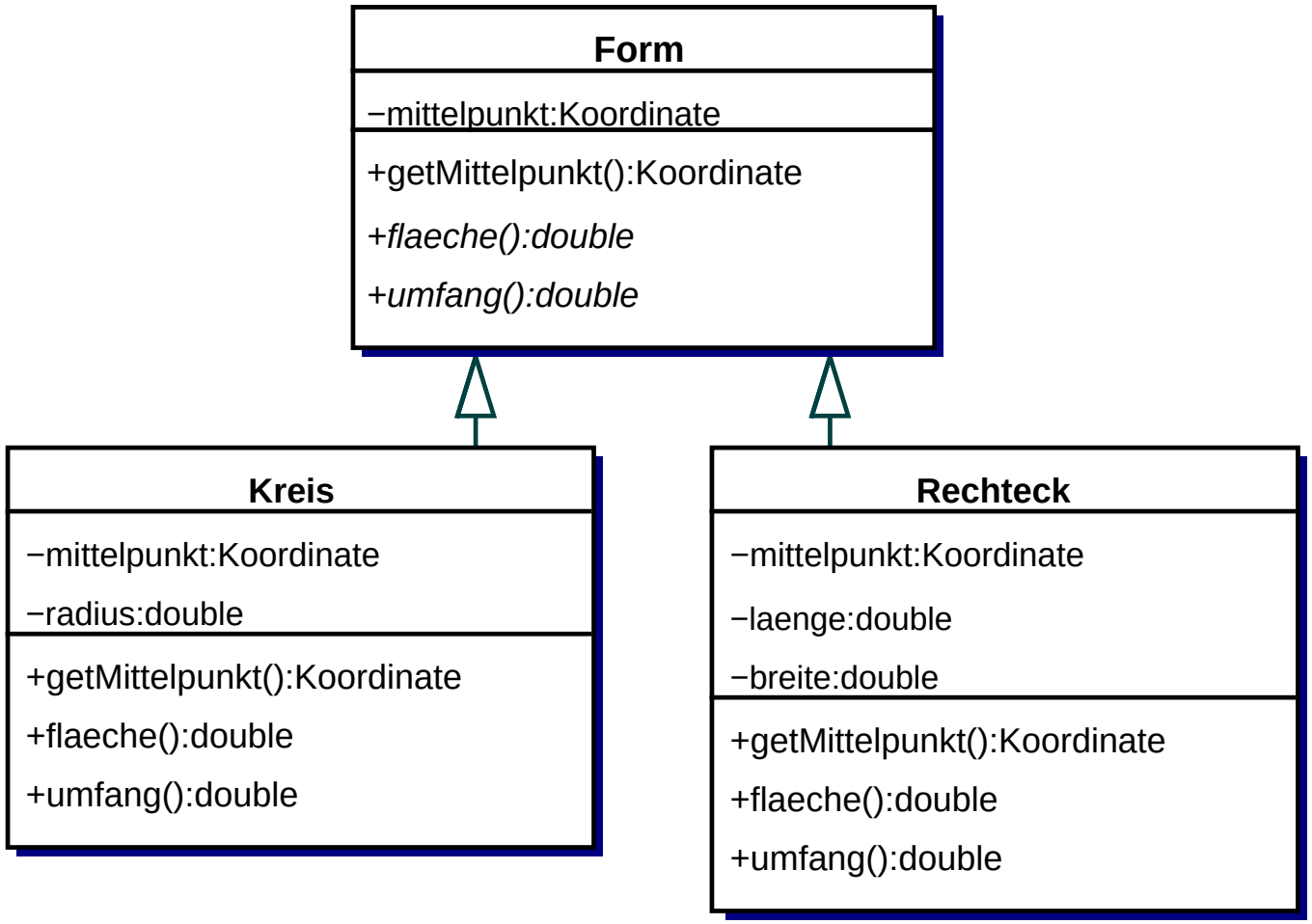
Abstraktion in der Programmierung:

- Häufig liegen in einer Klasse noch nicht genügend Details vor, um
  - für eine konkrete Realisierung von Methoden zu sorgen
    abstrakte Methoden
  - Objekte in einen sinnvollen Anfangszustand zu bringen
    abstrakte Klassen
  - alle nötigen Attribute und Assoziationen zu benennen
    Schnittstellen
- **Abstraktion** ist ein Hilfsmittel, um dennoch Festlegungen für speziellere Klassen treffen zu können, in denen die Details bekannt sind

Beispiel für abstrakte Klasse:



Beispiel für abstrakte Methode:



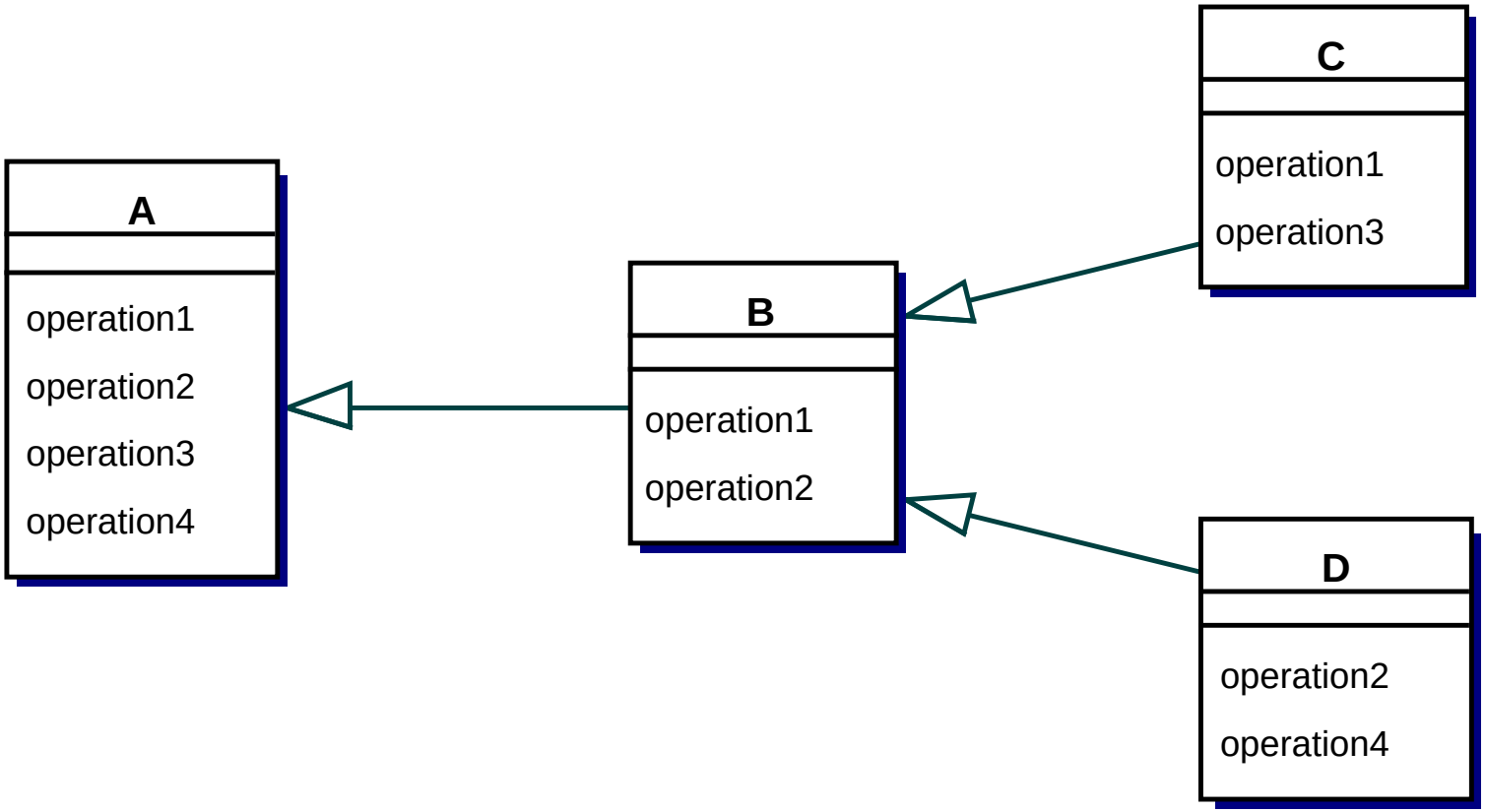
Vererbung:

- Vererbung ist programmiersprachliche Umsetzung der Spezialisierung
- Eigenschaften der Oberklasse werden an Unterklasse weitergegeben
- überall, wo Instanz der Oberklasse eingesetzt werden kann, kann auch Instanz der Unterklasse eingesetzt werden: **Substituierbarkeit**
- Vererbungsbeziehung besteht zwischen Klassen, nicht Objekten
- Bei Methoden gibt es verschiedene Arten der Vererbung
  - **Schnittstellenvererbung**: lediglich Signatur bzw. Kopf der Methode vererbt, für Realisierung ist Klasse selbst verantwortlich
  - **Implementierungsvererbung**: gesamte Methodendefinition (Kopf + Rumpf) wird vererbt

Spezialisierung von Methoden:

- Kontext: Eine Klasse, die durch Implementierungsvererbung die Methodendefinition ihrer Oberklasse erbt
- Problem: Die Klasse verfügt über spezielle Eigenheiten, so dass die geerbte Methodendefinition nur eingeschränkt nützlich, unbrauchbar oder optimierbar ist.
- Lösung: Die Klasse sorgt für eine eigene Definition der Methode, die die geerbte Methode redefiniert und das gewünschte, speziellere Verhalten realisiert. Dieses Vorgehen nennt sich **Überschreiben**.

Beispiel für Überschreiben:



Spezialisierung von Methoden (Forts.):

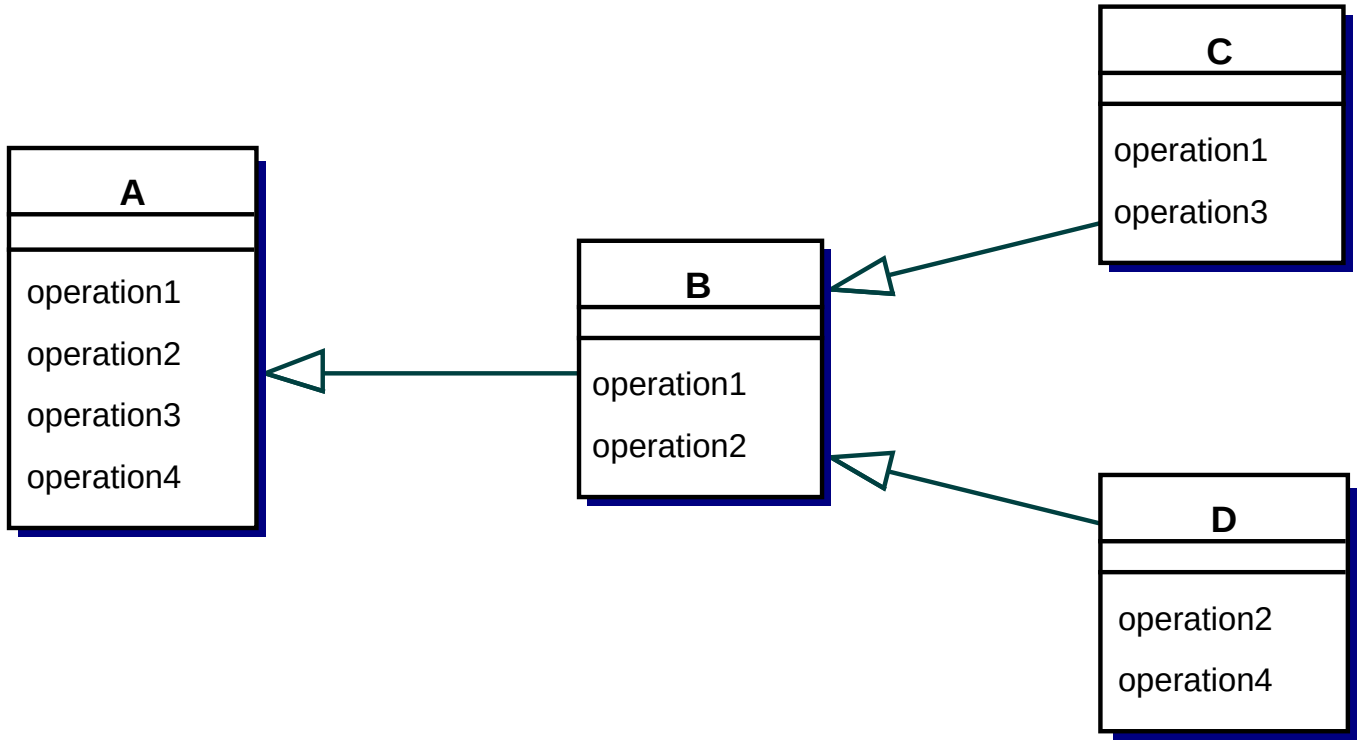
- Randbedingung: Die geerbte Methode muss überschreibbar sein!
  - manche Programmiersprachen ermöglichen es, Methoden als nicht überschreibbar zu definieren (z.B. Methoden mit Schlüsselwort `final` in Java)
  - in anderen Sprachen werden Methoden explizit als überschreibbar gekennzeichnet (z.B. virtuelle Methoden in C++).
- Variation der Methode beim Überschreiben
  - Methode der Unterklasse muss exakt gleiche Signatur haben, z.B. Java
  - in manchen Sprachen kann die Signatur einer Methode bei der Überschreibung verändert werden, z.B. Eiffel

Polymorphismus:

- Konkretisierung abstrakter Methoden und Überschreiben führen dazu, dass verschiedene Methodendefinitionen in Klassen einer Hierarchie vorliegen
- Wir deklarieren nun eine Variable mit einer Oberklasse/Schnittstelle als Typ und rufen eine Methode mit dem Variablennamen auf. Was passiert?
- Je nachdem, welcher Klasse das Objekt angehört, das der Variable zugewiesen ist, kann unterschiedliches Verhalten entstehen!
  Dies nennt man **Polymorphismus**.

Zurück zu unserem Überschreibungsbeispiel:

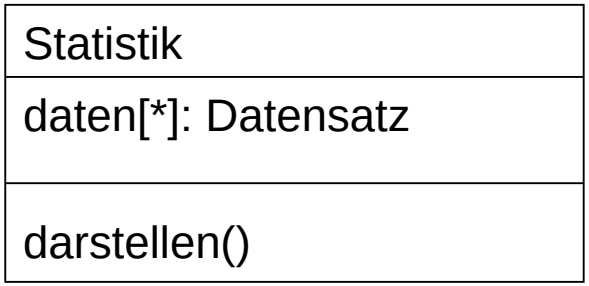
```
A obj;
obj = new C();
obj.op1();
obj.op2();
obj.op3();
obj.op4();
obj = new D();
obj.op1();
obj.op2();
obj.op3();
obj.op4();
```

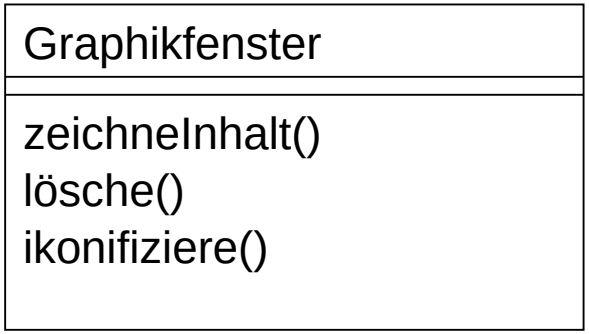


Wiederverwendung von Objekteigenschaften:

- Vererbung: neue Klasse erbt Eigenschaften der Klasse, die wiederverwendet werden soll
  - eventuell wollen wir aber nur Teile der Eigenschaften benutzen
  - jede Änderung der wiederverwendeten Klasse verändert auch die neue Klasse
- Delegation: Objekte der neuen Klasse „kennen“ Objekt der wiederzuverwendenden Klasse und beauftragen dieses mit der Bearbeitung
  - genaue Auswahl möglich, welche Eigenschaften wiederverwendet werden sollen
  - Austausch des Objektes zur Laufzeit möglich

Beispiel zur Wiederverwendung:





Beispiel zur Wiederverwendung:

Beispiel zur Wiederverwendung: