Kapitel 6

Grundlagen der objektorientierten Programmierung

Skript zur Vorlesung
Einführung in die Programmierung
im Wintersemester 2012/13
Ludwig-Maximilians-Universität München
(c) Peer Kröger, Arthur Zimek 2009, 2012

Übersicht

6.1 Einführung

6.2 Entwurf objektorientierter Programme

6.3 Klassen und Objekte in Java

6.4 Zusammenspiel von imperativen und objektorientierten Aspekten in Java

Übersicht

6.1 Einführung

6.2 Entwurf objektorientierter Programme

6.3 Klassen und Objekte in Java

6.4 Zusammenspiel von imperativen und objektorientierten Aspekten in Java

6.1 Einführung

- Elementarer Bestandteil der Algorithmen-Entwicklung ist die Modellierung eines Ausschnitts aus der realen Welt.
- Bisherige Modellierungsansätze:
  - Funktionale Programmierung:
    Die Lösung eines Problems wird als funktionaler (mathematischer) Zusammenhang zwischen Ein- und Ausgabedaten modelliert.
  - Imperative Programmierung:
    Die Lösung eines Problems wird als Menge von sequentiellen Abarbeitungsschritten modelliert.
- Beides sind wichtige Ansätze. Mit zunehmender Komplexität der Probleme (und damit der Ausschnitte der realen Welt, die man modellieren muss) führen diese Ansätze allerdings zu immer komplexeren und unübersichtlicheren Programmen.
- Der objektorientierte Ansatz bietet eine Lösung, komplexe Systeme in kleinere Komponenten zu zerlegen, die sich leichter beherrschen lassen

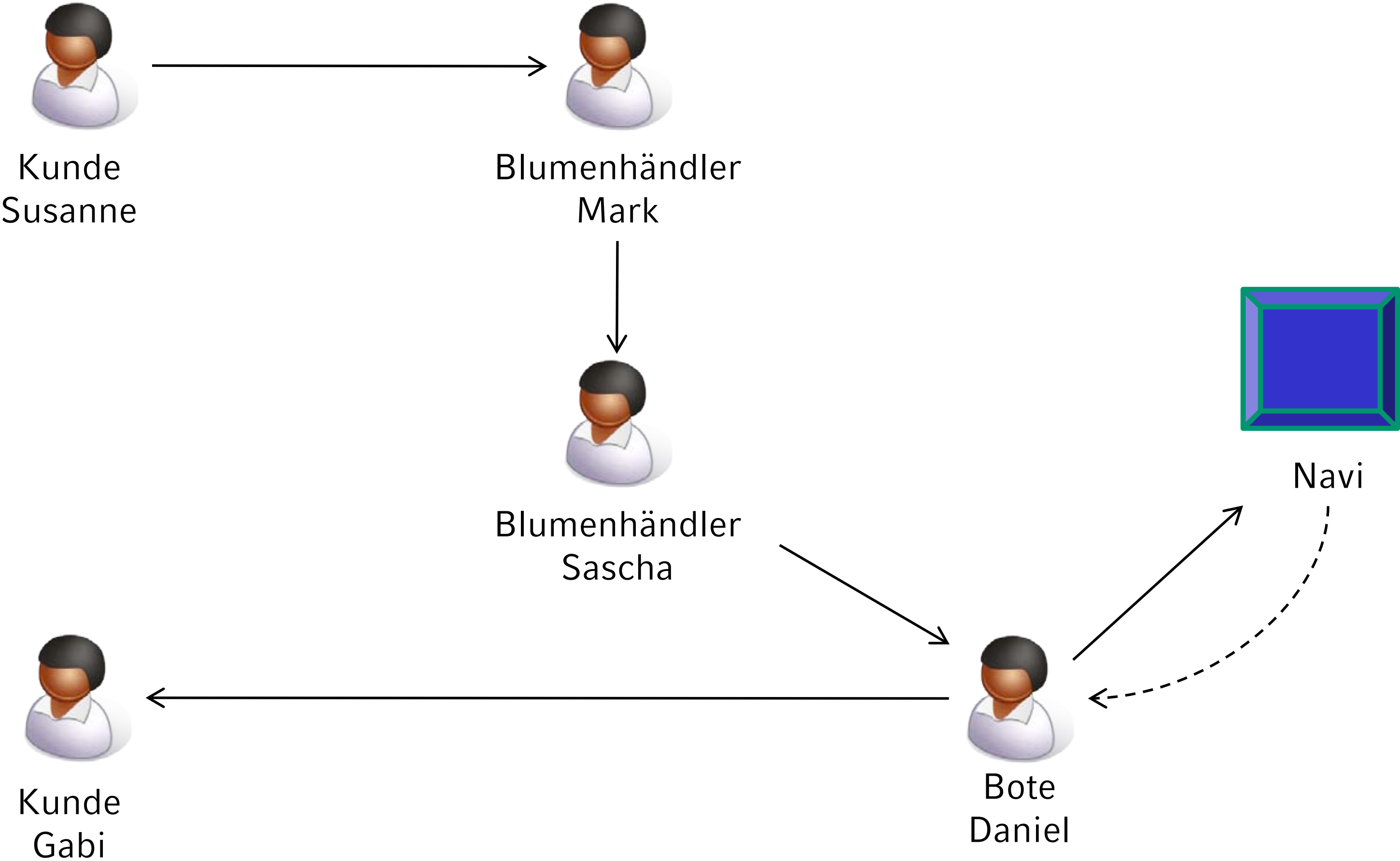
6.1 Einführung

- Objektorientierte (OO) Programmierung ist DAS Programmierparadigma der 90er Jahre.
- OO Programmierung ist ein weiterer Ansatz, Problemstellungen der realen Welt zu modellieren.
- Der OO Ansatz orientiert sich dabei an "Dingen" (Objekte, die gewisse Eigenschaften und ein gewisses Verhalten haben), die modelliert werden müssen.
- Genau genommen ist der OO Ansatz ein Konzept, komplexe Daten und Programmzustände zu modellieren. Die tatsächlichen Algorithmen werden meist mit einer Mischung aus funktionalen und imperativen Konzepten notiert.
- Beispiel Java: Die Algorithmen werden in der Regel mittels imperativer Konzepte implementiert, wogegen die Daten und deren Zustände OO modelliert sind.

6.1 Einführung

- Die OO Sichtweise stellt sich die Welt als *System von Objekten* vor, die untereinander *Botschaften* austauschen.
- Beispiel
  - Susanne in Rosenheim möchte ihrer Freundin Gabi in Buxtehude einen Blumenstrauß schicken.
  - Susanne geht daher zum Blumenhändler Mark und erteilt ihm den Auftrag
  - In der OO Programmierung ist Mark ein Objekt: Susanne sendet an Mark eine Botschaft: „Sende 7 gelbe Rosen an Gabi, Schneestr. 1 in Buxtehude"
  - Susanne hat getan, was sie konnte, nun ist es in Marks Verantwortung, den Auftrag zu bearbeiten
  - Mark versteht die Botschaft und weiß, was zu tun ist, d.h. er kennt einen Algorithmus für das verschicken von Blumen
  - Er versucht einen Blumenhändler in Buxtehude finden: er findet Florist Sascha und schickt ihm eine leicht veränderte Botschaft (z.B. mit Absender)
  - Mark ist damit fertig: er hat die Verantwortung für den Prozess an Sascha weitergegeben
  - Auch Sascha hat einen zur Botschaft passenden Algorithmus parat: er stellt die 7 Rosen zusammen und beauftragt seinen Boten Daniel, sie auszuliefern
  - Daniel muss den Weg zur Zieladresse finden und fragt sein Navi, das ihm entsprechend antwortet
  - Daniel findet den Weg, überreicht Gabi die Blumen und teilt ihr in einer Botschaft den Absender mit
  - Damit ist der von Susanne angestoßene Vorgang, an dem mehrere Objekte beteiligt waren, beendet

- Schematisch



6.1 Einführung

- Objekte besitzen *Eigenschaften* (*Attribute*)
  - Beispiel: ein Attribut eines Blumenhändlers ist der Ort an dem er sein Geschäft hat; weitere typische Attribute sind Name, Telefonnummer, Öffnungszeiten, Warenbestand, ...
- Objekte können bestimmte *Operationen* (*Methoden*) ausführen
  - Beispiel: ein Blumenhändler kann einen Lieferauftrag für Blumen entgegennehmen, Sträuße binden, Boten schicken, Blumen beim Großhandel einkaufen, ...
- Wenn ein Objekt eine geeignete Botschaft empfängt, wird eine zur Botschaft passende Operation gestartet (Methode aufgerufen)
- Der Umwelt (d.h. den anderen Objekten) ist bekannt welche Methoden ein Objekt beherrscht
- Allerdings weiß die Umwelt von den Methoden nur:
  - Was sie bewirken
  - Welche Daten sie als Eingabe benötigen

6.1 Einführung

- Die Umwelt weiß aber nicht, wie das Objekt funktioniert, d.h. nach welchen Algorithmen die Botschaften verarbeitet werden
- Das bleibt „privates“ Geheimnis des Objekts
  - Z.B. hat Susanne keine Ahnung, wie Mark den Blumentransport bewerkstelligt (es interessiert sie vermutlich auch gar nicht)
  - Susannes Aufgabe war einzig und allein, ein für ihr Problem *geeignetes* Objekt zu finden und ihm eine geeignete Botschaft zu senden; ungeeignete Objekte wären z.B. Kati die Zahnärztin oder Markus der Immobilienmakler gewesen, denn diese Objekte hätten Susannes Nachricht gar nicht verstanden
  - Für Susanne war zudem wichtig, zu wissen, *wie* sie die Botschaft für Mark formulieren muss
- Eine Methode ist die Implementierung eines Algorithmus, in Java kommt hier das imperative Paradigma ins Spiel

6.1 Einführung

- Die Objekte in diesem Beispiel kann man in *Gruppen* (*Klassen*) einteilen
  - Sascha und Mark sind Blumenhändler:
    - Sie beherrschen *die selben Methoden* und besitzen *die selben Attribute* (z.B. Ort oder Öffnungszeiten)
    - Die Attribute haben aber *unterschiedliche Werte*
  - Man sagt: Sascha und Mark sind Objekte (*Instanzen*) der Klasse „Blumenhändler“
- Eine Klasse ist eine Definition eines bestimmten Typs von Objekten, ein Bauplan indem die Methoden und Attribute beschrieben werden
  - Nach diesem Schema können Objekte (Instanzen) einer Klasse erzeugt werden
  - Ein Objekt ist eine Konkretisierung (Inkarnation) einer Klasse
  - Alle Instanzen einer Klasse sind von der Struktur (Methoden/Attribute) her gleich, sie unterscheiden sich allein in der Belegung ihrer Attribute mit Werten
    - Beispiel: Sascha und Mark haben beide das Attribut Ort, aber bei Sascha hat es den Wert „Buxtehude“ und bei Mark den Wert „Rosenheim“

6.1 Einführung

- Im Prinzip ist eine Klasse eine Sorte (Wertebereich)
- Die Objekte der Klasse sind die „Literale“ also Werte der Sorte
- Andersrum kann auch eine Sorte wie **`int`** als eine Klasse von Objekten (eine Teilmenge der ganzen Zahlen) aufgefasst werden; das einzige Attribut ist der Wert, der durch das Literal dargestellt wird
- Jedes Objekt (Literal) aus **`int`** unterscheidet sich durch ein den Wert dieses Attributs
- Die „Klasse“ stellt verschiedene „Methoden“ bereit, wie z.B. „+“, „==„, etc., die von allen Objekten „beherrscht“ werden
- Klassen heißen auch *benutzereigene* (*abstrakte*) *Datentypen*, da sie Mengen von Objekten mit Funktionalitäten zur Verfügung stellen

6.1 Einführung

Allgemein:

- Eine *Klasse* definiert die *Attribute* und *Methoden* ihrer Objekte.
- Der *Zustand* eines konkreten Objekts wird durch seine Attributwerte und *Verbindungen* (*Links*) zu anderen konkreten Objekten bestimmt.
- Das mögliche *Verhalten* eines Objekts wird durch die Menge von Methoden beschrieben.
- Die wichtigsten Konzepte der OO Programmierung sind:
  - Abstraktion
  - Kapselung
  - Wiederverwendung
  - Beziehungen
  - Polymorphismus

  die wir uns im Detail genauer anschauen:

Abstraktion

- Jedes erzeugte Objekt einer Klasse hat seine eigene Identität.
- Beispiel: Lichtschalter in einem Haus
  - Alle Lichtschalter sind gleich zu bedienen.
  - Alle Lichtschalter sind gleich konstruiert.
  - Dennoch unterscheidet sich der Lichtschalter für die Flurbeleuchtung vom Lichtschalter fürs Badezimmer: Es ist nicht derselbe Lichtschalter.
- Statt jeden einzelnen Lichtschalter neu zu modellieren/programmieren, abstrahiert man eine Klasse „Lichtschalter“, der alle Funktionalitäten nur einmal (für alle möglichen Lichtschalter) implementiert
- Abstraktion hilft, Details zu ignorieren, und reduziert damit die Komplexität des Problems. Dadurch werden komplexe Apparate und Techniken beherrschbar.

Kapselung

- Eine Klasse ist die Zusammenfassung einer Menge von Daten (Attributen, auch: *Membervariablen* oder *Instanzvariablen*) und darauf operierender Funktionen (Methoden).
- Die Attribute werden für jedes konkrete Objekt neu angelegt bzw. belegt. Sie repräsentieren den Zustand der Objekte.
- Die Methoden sind nur einmal realisiert / definiert und operieren bei jedem Aufruf auf den Daten eines bestimmten konkreten Objekts. Sie repräsentieren das Verhalten der Objekte.
- Kapselung bedeutet, dass (von gewollten Ausnahmen abgesehen) die Methoden die einzige Möglichkeit darstellen, mit einem konkreten Objekt zu kommunizieren und so Informationen über dessen Zustand zu gewinnen oder diesen zu verändern.

6.1 Einführung

- Kapselung hilft, die Komplexität der Bedienung eines Objekts zu reduzieren.
- Durch Kapselung werden die Implementierungsdetails von Objekten verborgen.
- Dadurch können Daten nicht bewusst oder versehentlich verändert werden.
- Kapselung ist daher auch ein wichtiger Sicherheitsaspekt: Ein direkter Zugriff auf die Daten wird unterbunden, der Zugriff erfolgt nur über definierte *Schnittstellen*, die bereitgestellten Methoden.
- Beispiel: welche Attribute den Zustand eines Lichtschalters definieren kann uns egal sein, solange wir wissen, wie wir den Lichtschalter bedienen müssen; tatsächlich können wir den Zustand eines Lichtschalters nur über dessen „Schnittstelle“ verändern (indem wir den Schalter drücken oder drehen)

Wiederverwendbarkeit

- Durch Abstraktion und Kapselung wird die Wiederverwendbarkeit von Programmteilen gefördert.
- Beispiel: Sog. Collections sind Objekte, die Sammlungen anderer Objekte aufnehmen und verarbeiten können.
  - Collections sind meist sehr kompliziert aufgebaut.
  - Collections haben meist eine einfache Art der Bedienung (Schnittstelle).
  - Werden Collections als Klasse realisiert, werden durch die Kapselung die komplexen Details des Aufbaus “wegabstrahiert”.
  - Dies erleichtert die Wiederverwendung: Wenn ein Programm eine spezielle Collection benötigt, muss ein Objekt der passenden Klasse erzeugt werden. Das Programm kann über die einfache Schnittstelle (Methoden der Klasse) darauf zugreifen.
- Wiederverwendbarkeit hilft, effizienter und fehlerfreier zu programmieren.

Beziehungen

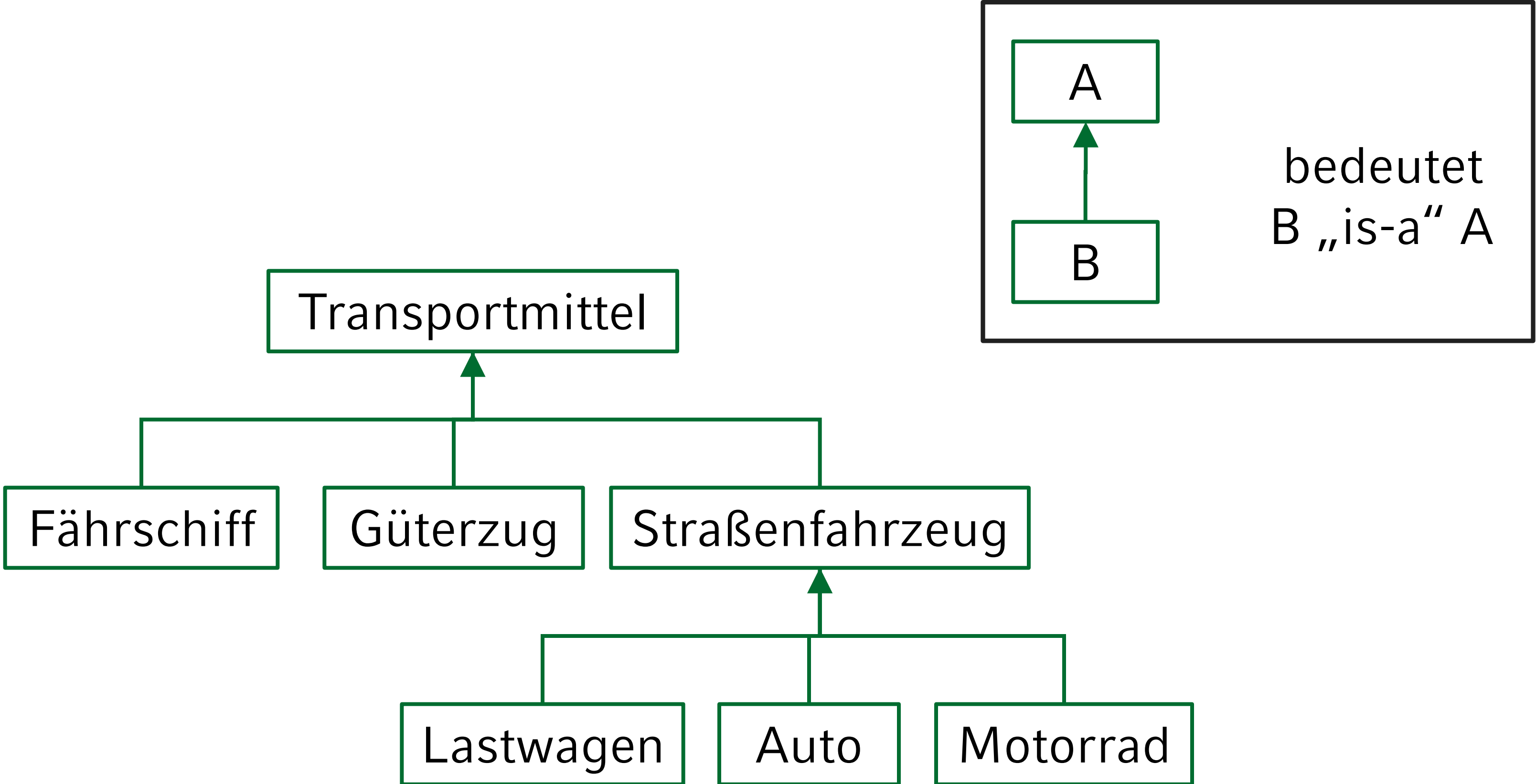
- Klassen und Objekte existieren typischerweise nicht isoliert, sondern stehen in Beziehung zueinander.
- Beispiel:
  - Sowohl ein Motorrad, als auch ein Auto und ein Lastwagen sind Straßenfahrzeuge.
  - Ein Lastwagen kann möglicherweise Motorräder aufnehmen.
  - Ein Fährschiff ist ebenfalls ein Transportmittel und kann viele Autos, Lastwagen und Motorräder aufnehmen.
- Grundsätzlich gibt es in der OO Programmierung drei Arten von Beziehungen:
  - Generalisierung und Spezialisierung (“is-a”-Beziehungen),
  - Aggregation und Komposition (“part-of”-Beziehungen),
  - Verwendungs- und Aufrufbeziehungen.

6.1 Einführung

Generalisierung und Spezialisierung

- Eine "is-a"-Beziehung zwischen zwei Klassen *A* und *B* sagt aus, dass "*B* ein *A* ist", also *B* alle Eigenschaften von *A* besitzt (und darüberhinaus typischerweise noch ein paar mehr).
- *B* ist eine *Spezialisierung* von *A*, es hat mehr Eigenschaften.
- *A* ist eine *Generalisierung* von *B*, es hat weniger Eigenschaften.
- Beispiel:
  - Motorrad, Auto und Lastwagen sind zwar paarweise unterschiedlich, aber alle sind Straßenfahrzeuge.
  - Straßenfahrzeuge sind, genau wie Fährschiffe (und Güterzüge etc.), Transportmittel.
  - Die entsprechende "is-a"-Beziehung ist in der *Hierarchie* auf der folgenden Folie veranschaulicht.

6.1 Einführung



6.1 Einführung

- "is-a"-Beziehungen werden in der OO Programmierung durch *Vererbung* modelliert.
- Die speziellere Klasse wird dabei nicht komplett neu definiert, sondern von der allgemeineren Klasse *abgeleitet*.
- Die speziellere Klasse *erbt* implizit alle Eigenschaften der allgemeineren Klasse (auch: *Vaterklasse*) ohne, das sie nochmal explizit aufgeführt werden müssen. Eigene Eigenschaften können nach Belieben hinzugefügt werden.
- Vererbungen können mehrstufig sein, d.h. eine abgeleitete Klasse kann wiederum Vaterklasse für andere abgeleitete Klassen sein (siehe *Vererbungshierarchie* auf der vorherigen Folie).
- Grundsätzlich kann eine Klasse auch von mehreren Vaterklassen abgeleitet sein. Beispielsweise ist ein Amphibienfahrzeug eine Spezialisierung sowohl eines Wasserfahrzeug als auch eines Landfahrzeug. In diesem Fall spricht man von *Mehrfachvererbung*.

6.1 Einführung

Aggregation und Komposition

- Eine "part-of"-Beziehung beschreibt die *Zusammensetzung* eines Objekts aus anderen Objekten.
- Ist diese Zusammensetzung essentiell für die Existenz des zusammengesetzten Objekts, spricht man von *Komposition*. Ein Güterzug besteht z.B. aus einer Lokomotive und mehreren Anhängern. Ohne diese Teile gibt es keine Güterzüge.
- Ist die Zusammensetzung nicht essentiell für die Existenz des zusammengesetzten Objekts, spricht man von *Aggregation*. Zwischen Straßenfahrzeug und Fährschiff besteht eine "part-of"-Beziehung, aber das Fährschiff kann auch leer fahren.
- **Bemerkung**: In den meisten OO Programmiersprachen werden Komposition und Aggregation gleich behandelt.

6.1 Einführung

Verwendungs- und Aufrufbeziehungen

- Verwendungs- und Aufrufbeziehungen (*Assoziationen*) kommen dadurch zustande, dass z.B.
  - eine Methode einer Klasse *A* während ihrer Ausführung ein temporäres Objekt der Klasse *B* benutzt,
  - in der Parameterliste einer Methode einer Klasse *A* eine Variable definiert wird, die als Typ die Klasse *B* hat,
  - etc.

Polymorphismus

- Betrachten wir noch einmal Fährschiffe: Sie können Autos, Lastwägen und Motorräder – oder ganz allgemein Straßenfahrzeuge – aufnehmen.
- Um diese Aggregation zu beschreiben, genügt es also zu definieren, dass Fährschiffe Straßenfahrzeuge aufnehmen.
- *Polymorphismus* besagt, dass nicht nur Objekte der Klasse Straßenfahrzeuge aufgenommen werden können, sondern auch Objekte aller abgeleiteten Klassen von der Vaterklasse Straßenfahrzeuge, also auch Autos, Lastwägen und Motorräder.
- Die zusätzlichen Eigenschaften der abgeleiteten Klassen sind für das Fährschiff irrelevant, wichtig ist, dass sie die Eigenschaften eines Straßenfahrzeugs haben (was sie als Objekte einer von Straßenfahrzeuge abgeleitete Klasse auch haben).
- Andersherum funktioniert Polymorphismus nicht!

6.1 Einführung

- Ein weiterer Aspekt ist, dass Objekte unterschiedlicher Klassen die gleiche Funktionalität besitzen können (die allerdings in jeder Klasse unterschiedlich realisiert ist).
- Beispiel:
  - Bei allen Straßenfahrzeugen gibt es die Funktionalität, die Anzahl der Reifen abzufragen (realisiert durch eine Methode "anzahlReifen").
  - Diese Methode kann in allen drei Klassen (Auto, Lastwagen, Motorrad) denselben Namen haben. In allen drei Fällen steckt aber ein unterschiedlicher Algorithmus dahinter.
  - Diese Funktionalität kann bereits in der Vaterklasse zur Verfügung stehen und in den abgeleiteten Klassen *überschrieben / redefiniert* werden.
  - Wird im Zusammenhang eines Fährschiffes die Anzahl der Reifen eines Straßenfahrzeugs benötigt, das auf diesem Schiff gerade transportiert wird, wird dynamisch ausgewählt, welche Methode ausgeführt wird, ohne dass sich das Fährschiff darum kümmern muss (ist das Straßenfahrzeug z.B. ein Auto, wird "anzahlReifen" der Klasse Auto ausgeführt).

Übersicht

6.1 Einführung

6.2 Entwurf objektorientierter Programme

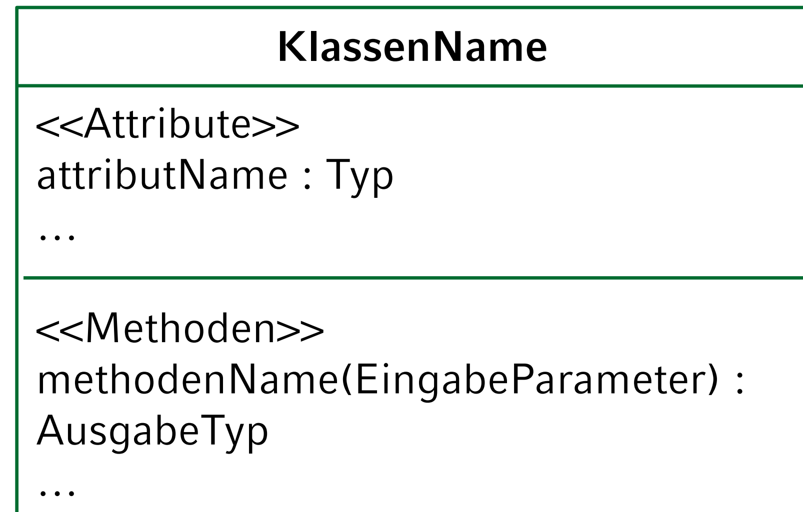
6.3 Klassen und Objekte in Java

6.4 Zusammenspiel von imperativen und objektorientierten Aspekten in Java

6.2 Entwurf objektorientierter Programme

- Wie schon bei der imperativen oder funktionalen Programmierung ist der Entwurf der eigentlichen Algorithmen (und damit die Modellierung der Problemstellung) die entscheidende Herausforderung.
- Für den OO Entwurf gibt es die *Unified Modelling Language* (*UML*).
  - UML ist eine Konzeptsprache zur Modellierung von Software-Systemen (insbesondere, aber nicht zwingend: Objektorientierter Programme).
  - UML ist eine Art Pseudo-Code, der allerdings eine wohl-definierte Semantik besitzt und von vielen Programmen verarbeitet werden kann (z.B. können Implementierungsdetails z.B. in Java-Notation angegeben werden aus denen automatisch Java-Code generiert werden kann.
  - UML-Code selbst ist nicht ausführbar. Dennoch wird UML von vielen Experten als Prototyp für die nächste Generation von Programmiersprachen betrachtet.
  - Im Rahmen dieser Vorlesung werden wir UML-*Klassendiagramme* verwenden, die die statische Struktur eines Programms im Wesentlichen durch Klassen, Objekte und deren Beziehungen zu einander konzeptionell beschreiben. Realisierungsdetails werden meist mit anderen Diagrammtypen beschrieben.
  - Einen tieferen Einblick in UML erhalten Sie in den Vorlesungen zur Software-Entwicklung.

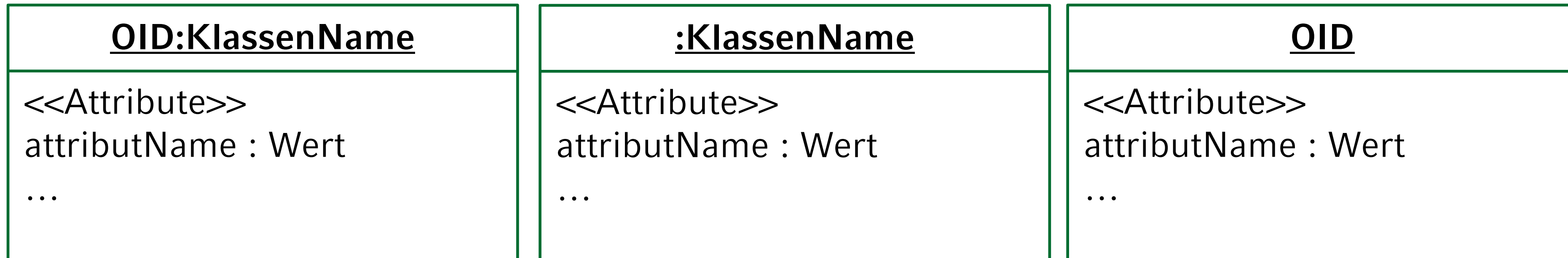
- Klassen
  - Wie bereits erwähnt setzen die Konzepte der Klassen und Objekte die OO Grundkonzepte "Abstraktion" und "Kapselung" um.
  - Klassen modellieren die gemeinsamen Eigenschaften von Objekten, insbesondere deren Attribute und Methoden.
  - Attribute beschreiben den Zustand von Objekten einer Klasse und sollten für andere Benutzer (z.B. Objekte) verborgen (nicht sichtbar) sein.
  - Methoden beschreiben das Verhalten der Objekte einer Klasse und sollten für andere Benutzer bekannt (sichtbar) und verfügbar sein.
- Klassendefinition in UML:



6.2 Entwurf objektorientierter Programme

- Die *Sichtbarkeit* von Methoden und Attributen muss explizit spezifiziert sein und sollt die Konzepte Kapselung und Abstraktion (wie auf der vorherigen Folie diskutiert) berücksichtigen.
- Die Symbole zur Spezifikation der Sichtbarkeit von Klassen und deren Elementen sind unter anderem:
  - + bzw. Public: Das entsprechende Element (Attribut / Methode) ist von außen (z.B. Objekten anderer Klassen) sichtbar.
  - - bzw. Private: Das entsprechende Element (Attribut / Methode) ist von außen (z.B. Objekten anderer Klassen) NICHT sichtbar.
- Darüberhinaus gibt es weitere Möglichkeiten, die Sichtbarkeit von Elementen einer Klasse zu spezifizieren. Diese lernen wir später kennen.
- *Achtung*: Es gibt in UML (anders als z.B. in Java) keine Default-Spezifikation für Elemente einer Klasse, d.h. deren Sichtbarkeit muss immer explizit angegeben sein!

- Konkrete Objekte einer Klasse werden in UML mit den folgenden drei Alternativen dargestellt:

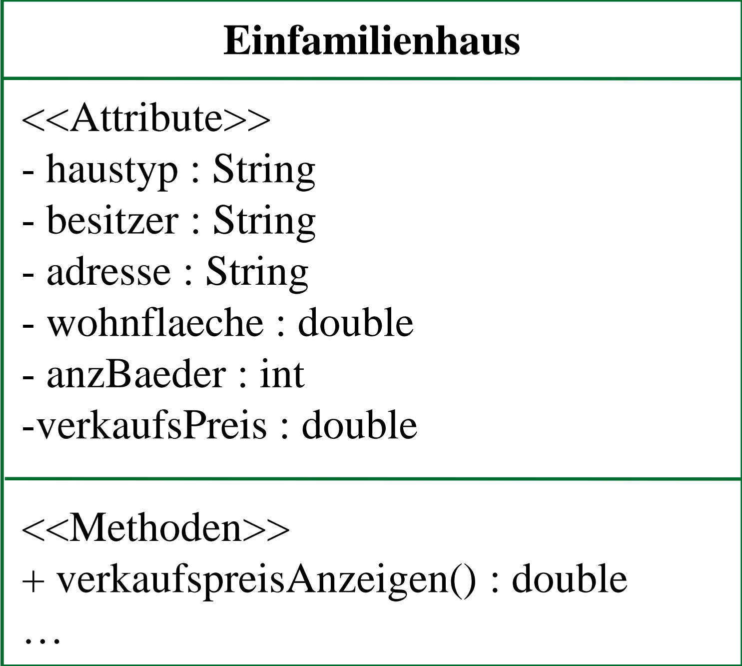


(wie gesagt, alle drei Varianten sind Darstellungsmöglichkeiten für Objekte in UML)

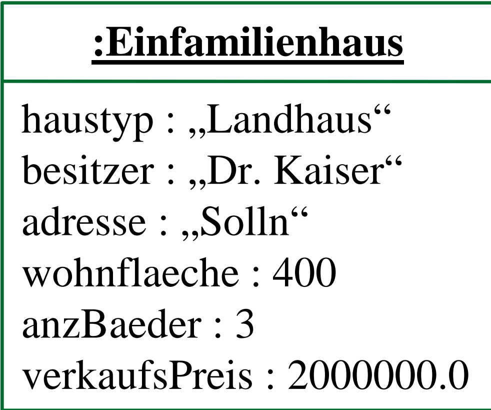
6.2 Entwurf objektorientierter Programme

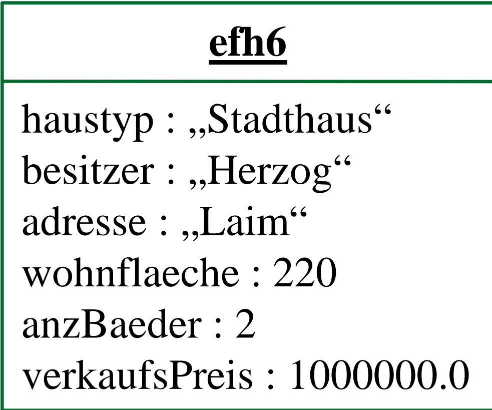
- Beispiel:
  Definition einer Klasse „Einfamilienhaus“ und drei konkrete Objekte

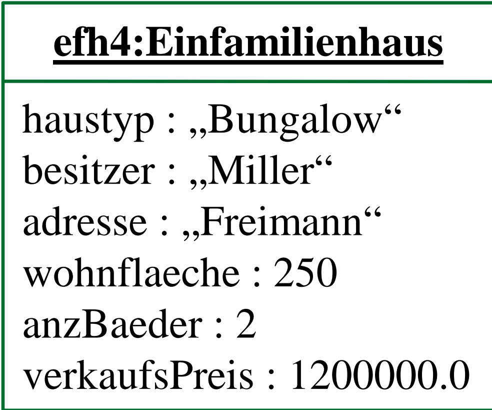
Klasse



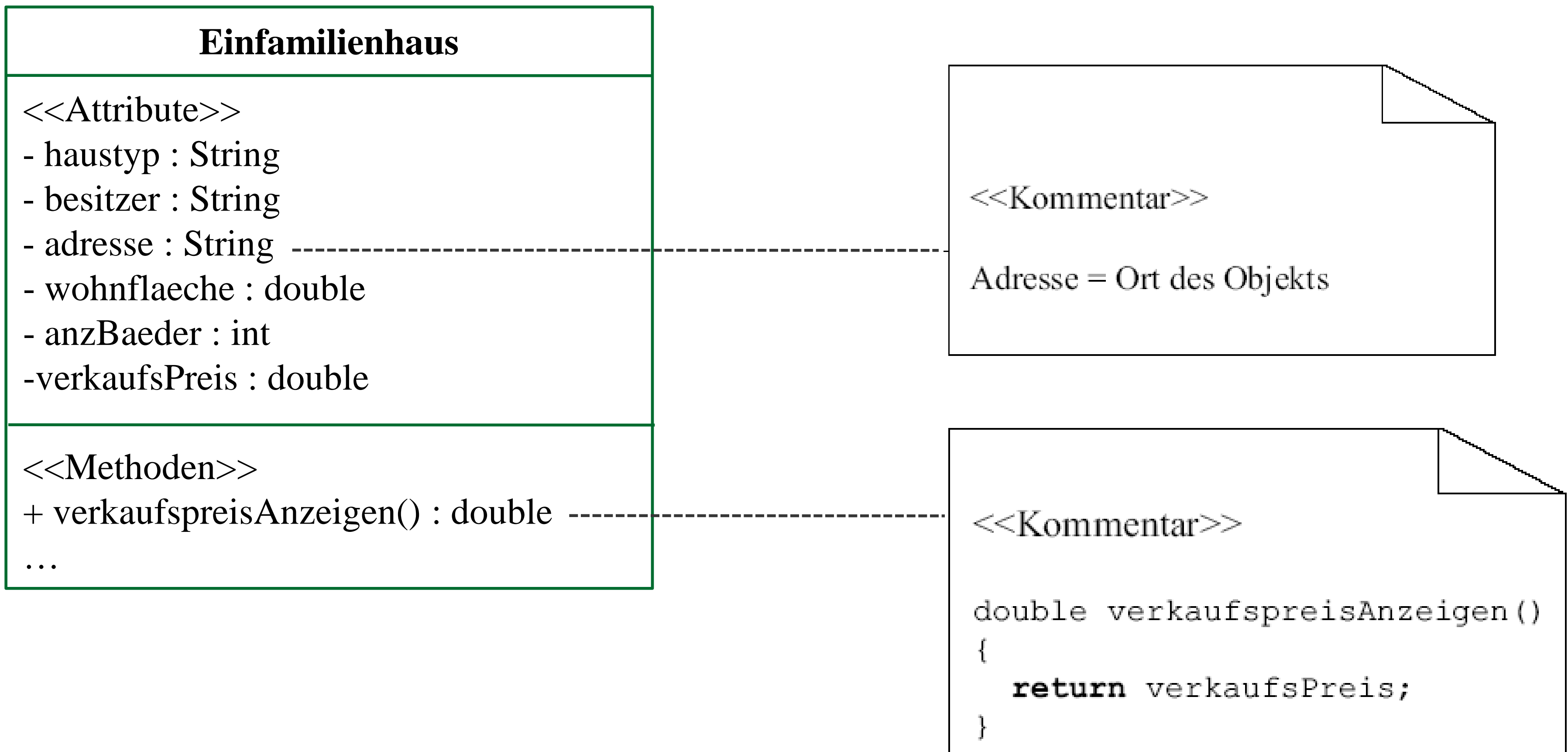
Objekte







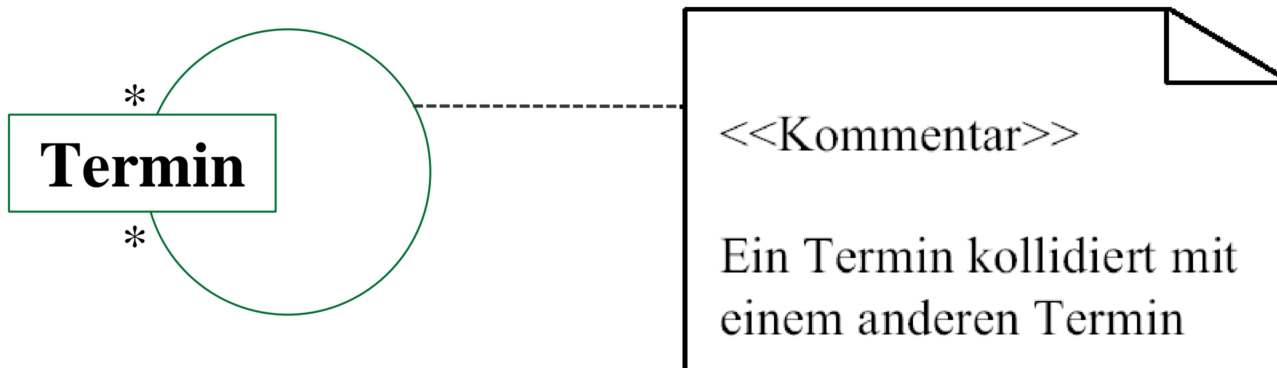
- Kommentare werden in UML wie folgt dargestellt:



- Verwendungsbeziehungen
  - Verwendungsbeziehungen sind die allgemeinste Form von Assoziationen zwischen Objekten verschiedener Klassen.
  - Die Situation, dass Objekte der Klasse A Objekte der Klasse B verwenden und umgekehrt, stellt man in UML wie folgt dar:

  - *mult* bezeichnet die *Multiplizität* der Assoziation und steht u.a. für
    - * : beliebig viele,
    - *n..m* : mindestens n, maximal m,
    - Zusatz {*unique*} :Die verwendeten Objekte sind alle paarweise verschieden,
    - Zusatz {*ordered*} : Die verwendeten Objekte sind geordnet (impliziert {*unique*}).
  - Die Assoziation kann auch gerichtet werden, z.B. wenn nur Objekte der Klasse B Objekte der Klasse A verwenden:

- Beispiele



6.2 Entwurf objektorientierter Programme

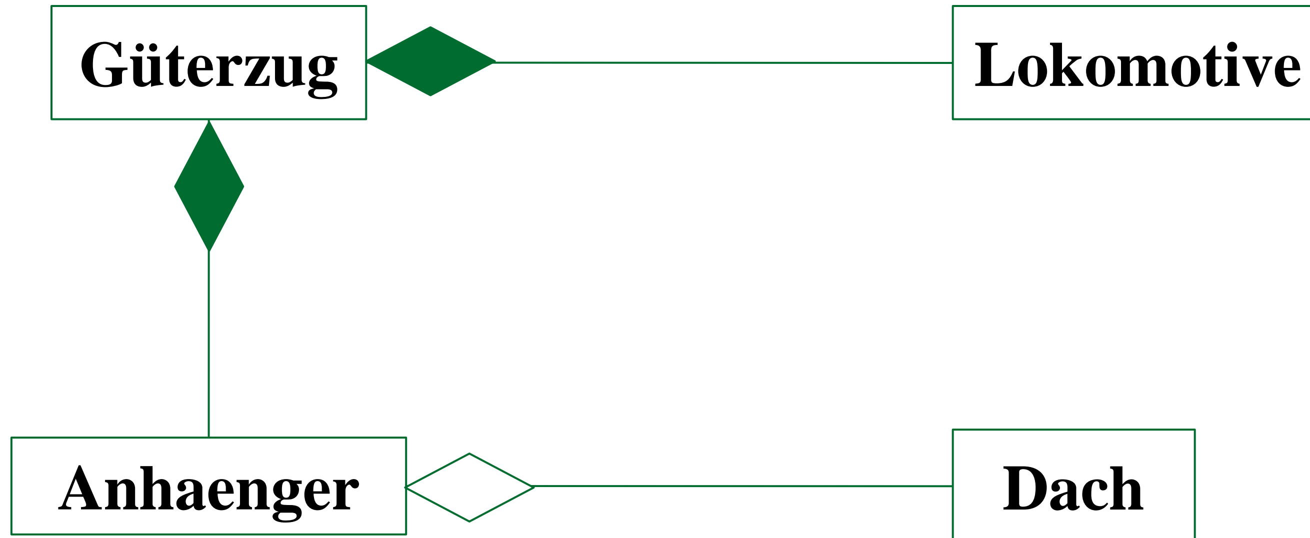
- Aggregation

- Komposition

- Generalisierung/Spezialisierung (Vererbung)

Hier können überall auch wieder Multiplizitäten angegebene werden

- Beispiele für Aggregation und Komposition:



- Beispiele für Aggregation und Komposition und Vererbung: