Modul: Programmierung B-PRG
Grundlagen der Programmierung 1

V17 UML in Grundzügen

Prof. Dr. Detlef Krömker
Professur für Graphische Datenverarbeitung
Institut für Informatik
Fachbereich Informatik und Mathematik (12)

Unsere heutigen Lernziele

Elemente des Systementwurfs kennenlernen

UML in elementaren Grundzügen begreifen:
- *Use Cases (Anwendungsfälle)*
- *Klassendiagramme*

und hierzu die grafische Notationen kennenlernen

Übersicht (1)

- **Einordnung der Entwurfsmethoden**
  - Rückblick Objektorientierung und deren Realisierung in Python
- **Übersicht zur Analyse**
  - Systemidee und Zielsetzung
  - Stakeholder
  - Systemkontext
  - Nichtfunktionale Anforderungen identifizieren und modellieren
  - Anwendungsfälle identifizieren und modellieren
  - Glossar erstellen
  - Fachklassen modellieren
- **Übersicht zum Design**
  - Prinzipielles Vorgehen
  - Entwicklung eines Klassenmodells
  - Klassendiagramm

Begriff Vorgehensmodelle im SWE

- *allgemein organisieren einen Prozess in verschiedene, strukturierte Abschnitte, denen wiederum entsprechende Methoden und Techniken der Organisation zugeordnet sind.*
- *Aufgabe eines Vorgehensmodells ist es, die allgemein in einem Gestaltungsprozess auftretenden Aufgabenstellungen und Aktivitäten in einer sinnfälligen logischen Ordnung darzustellen.*
- *Mit ihren Festlegungen sind Vorgehensmodelle organisatorische Hilfsmittel, die für konkrete Aufgabenstellungen (Projekte) individuell angepasst werden können und sollen … .*

aus Wikipedia: https://de.wikipedia.org/wiki/Vorgehensmodell

- Teil des Projektmanagements … behandeln wir am kommenden Montag

Einführung in UML: Unified Modeling Language

- Anders als die einfacheren Vorgänger vereinigt UML eine Vielzahl einzelner Notationen für verschiedene Aspekte aus dem Bereich des OO-Designs und es können deutlich mehr Details zum Ausdruck gebracht werden.
- Somit ist es üblich, sich auf eine Teilmenge von UML zu beschränken, die für das aktuelle Projekt ausreichend ist.
- Wir betrachten in PRG1 „nur“ das Wichtigste!

UML Werkzeuge

Wichtige typische Funktionalitäten:

- **Diagrammunterstützung:** Erzeugen und Bearbeiten von (standardgerechten) UML-Diagrammen: Kontrovers: Wie (wenn überhaupt) werden diese Diagramme aktualisiert?
- **Quelltexterzeugung:** das UML-Werkzeug fungiert als Codegenerator
- **Reverse Engineering:** das UML-Werkzeug erzeugt aus dem Quelltext als Eingabe die zugehörigen UML-Diagramme und Modelldaten.
- **„Roundtrip“-Engineering:** bedeutet, dass der Anwender die Möglichkeit hat, entweder die Modelldaten (durch Veränderung der entsprechenden Diagramme) oder den Quellcode zu verändern, und das Werkzeug das Gegenstück automatisch aktualisiert.

Für Sie interessant:

- das ArgoUML http://argouml.tigris.org/ zum Download (allerdings nur bis UML 1.4)
- Weitere Empfehlungen folgen.

Unified Modeling Language

- Ist mehr als "nur" die standardisierten Diagramme
- Ist eine **Modelierungssprache,** die im Systemdesign eingesetzt wird.
- UML ist prinzipiell unabhängig von Programmiersprachen und Betriebssystem.
- ➔ Konflikte im Detail mit Programmierkonventionen, z.B. Attribut-Namen:
  nach unseren PEP 8 Konventionen: important_number.
  streng nach UML 2.5: importantNumber
  "Einen Tot muss man sterben": Ich bevorzuge Namen nach PEP 8
- Aktuell wichtig sind die Vorgehensmodelle **Scrum, V-Modell, etc**.
- UML hat die Notationen des OOAD weitgehend konvergieren lassen.

Unser Vorgehen

- Orientiert sich am ***Object Engineering Process*** [Oestereich 1999]
- Dies ist eine spezielle Ausprägung des Unified Software Development Process [Jacobson 1999], siehe www.oose.de/oep
  beschrieben u.a. in Bernd Oestereich: Analyse und Design mit UML 2.3-Objektorientierte Softwareentwicklung, 9. Auflage, Oldenbourg 2009

  Alternative hierzu:
- **Rational Unified Process RUP** [Kruchten1998] mit diversen Werkzeugen von IBM/Rational unterstützt, siehe www.rational.com/rup

Übersicht (1)

- Einordnung der Entwurfsmethoden
  - Rückblick Objektorientierung und deren Realisierung in Python
- **Übersicht zur Analyse**
  - Systemidee und Zielsetzung
  - Stakeholder
  - Systemkontext
  - Nichtfunktionale Anforderungen identifizieren und modellieren
  - Anwendungsfälle identifizieren und modellieren
  - Glossar erstellen
  - Fachklassen modellieren

Systemidee und Zielsetzung

- In der Regel **zusammen mit dem Auftraggeber** entwickeln.
- Kurz ausformulieren: 5-20 Sätze.
- Geben Sie ihrem "Baby" (dem neuen Programm) einen Namen.
- Dazu gehört eine Auflistung der Features des Systems (Anzahl 5-15).
- Die Voraussetzungen müssen genannt werden (z.B. Hardware, Software, Entwicklungs- und Zielumgebung, Finanzen, Termine)
- Eine Abgrenzung: Was gehört **nicht** dazu.

Stakeholder

- Es gibt keine wirklich gute deutsche Übersetzung:
  Interessenvertreter. Interessenhalter, Anspruchsberechtigter
- Bewerten Sie die Wichtigkeit der Stakeholder anhand
  - ihrer Relevanz und
  - Risikos

  Wie groß ist der Aufwand, diesen Stakeholder zu berücksichtigen? (6 = vernachlässigbar, … , 1 = extrem hoch)

  Wie groß ist das Risiko, wenn der Stakeholder nicht berücksichtigt wird? (z.B. von 1=kein … 6 fatal)
- Identifiziere die Ansprechpartner

Use Cases

- Use Cases" dokumentieren während der Analyse die typischen Prozeduren **aus der Sicht der aktiven Teilnehmer (Akteure)** für ausgewählte Fälle.
- Akteure sind aktive Teilnehmer, die Prozesse in Gang setzen (initiieren) oder Prozesse am Laufen halten.

Use Case (Anwendungsfall)

- Akteure können sein:
  - Menschen, die direkt interaktiv mit dem System arbeiten oder
  - andere Systeme, die über Netzwerkverbindungen kommunizieren oder
  - interne Komponenten, die kontinuierlich laufen (wie beispielsweise die Uhr).
- "Use Cases" werden **informell dokumentiert** durch die Aufzählung einzelner Schritte, die zu einem Vorgang gehören und können in graphischer Form zusammengefasst werden, wo nur noch die Akteure, die zusammengefassten Prozeduren und Beziehungen zu sehen sind.

Natürlichsprachliche Beschreibung des Anwendungsfalls (nach Oestereich, OOAD)

Beschreibung Anwendungsfall	
Name	
Kurzbeschreibung	
Akteure	
Auslöser	
Ergebnis(se)	
Eingehende Daten	
Vorbedingungen	
Nachbedingungen	
Essentielle Schritte	
Offene Punkte	
Änderungshistorie	wann? wer? was?
Sonstiges, Anmerkungen	

Beispiel: Abläufe bei einer einfachen Bank-Anwendung

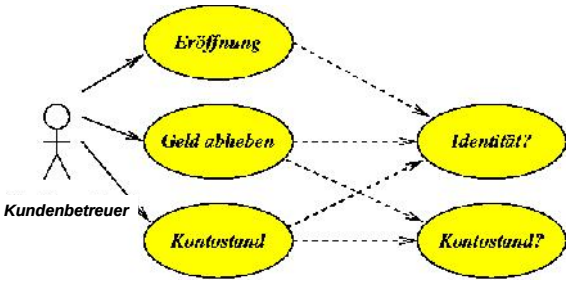
Aus welchen für die Nutzer (Akteur) sichtbaren Schritten bestehen einzelne typische Abläufe bei dem Umgang mit Bankkunden?

- Konto-Eröffnung
  - Feststellung der Identität
  - Persönliche Angaben erfassen
  - Kreditwürdigkeit überprüfen Geld abheben
- Geld abheben
  - Feststellung der Identität
  - Überprüfung des Kontostandes
  - Abbuchung des abgehobenen Betrages
- Auskunft über den Kontostand
  - Feststellung der Identität
  - Überprüfung des Kontostandes

Beispiel: Abläufe bei einer einfachen Bank-Anwendung

- Auf der nächsten Folie sind nur die Aktivitäten aufgeführt, die der Kundenbetreuer (Banker) im Umgang mit dem System ausübt.
- Der Akteur ist hier der Kundenbetreuer, weil er in diesen Fällen mit dem System arbeitet.
- Der Kunde würde nur dann zum Akteur, wenn er beispielsweise am Bankautomaten steht oder über das Internet auf sein Bankkonto zugreift.
- Interessant sind hier die Gemeinsamkeiten einiger Abläufe. So wird beispielsweise der Kontostand bei zwei Prozeduren überprüft.

Abläufe bei einer Bank-Anwendung

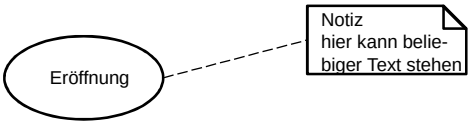


Notationen

- Eine **glatte Linie** mit einem Pfeil verbindet einen Akteur mit einem Prozedur (Anwendungsfall). Das bedeutet, dass die mit der Anwendungsfall verbundene Prozedur von diesem Akteur angestoßen bzw. durchgeführt wird.
- **Gestrichelte Linien** repräsentieren Beziehungen zwischen mehreren Prozeduren. Damit können Gemeinsamkeiten hervorgehoben werden.
- **Wichtig:** Pfeile repräsentieren **keine** Flussrichtungen von Daten. Es führt hier insbesondere kein Pfeil zu dem Kundenbetreuer zurück.

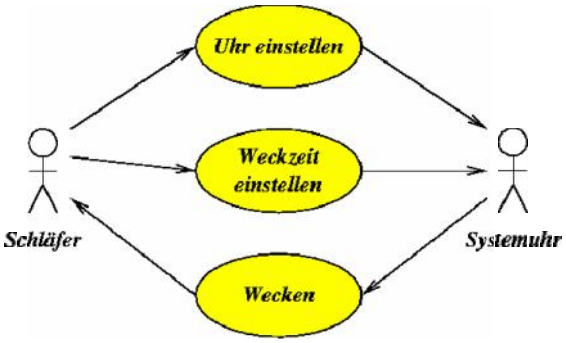
Notiz (Annotation, Kommentar, *note or comment)*

- **Notizen** werden durch **"Rechtecke mit einem Eselsohr"**, die einen Text enthalten, dargestellt..
- Diese Notizen werden durch eine **gestrichelte Linie** mit dem Modellelement verbunden.
- Kommentare können an beliebige UML-Modellelemente angefügt werden



Beachte: Notizen sind semantisch sehr schwache Elemente

Abläufe bei einem Wecker



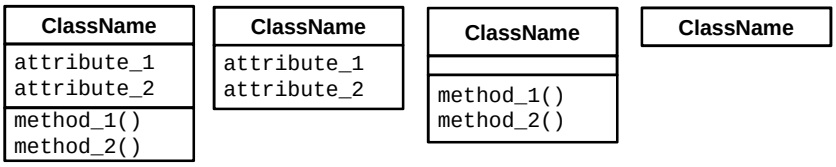
Beispiel Wecker

- Es können auch Pfeile von Prozeduren zu Akteuren gehen, wenn sie **eine Benachrichtigung** repräsentieren, die sofort wahrgenommen wird.
- Ein Wecker hat intern einen Akteur --- die Systemuhr. Sie aktualisiert laufend die Zeit und muss natürlich eine Neu-Einstellung der Zeit sofort erfahren.
- Das Auslösen des Wecksignals wird von der Systemuhr als Akteur vorgenommen. Diese Prozedur führt (hoffentlich) dazu, dass der Schläfer geweckt wird. In diesem Falle ist es berechtigt, auch einen Pfeil von einer Prozedur zu einem menschlichen Akteur zu ziehen.

OO-Design ➔ Klassenmodell (-diagramm)

- Ein **Klassenmodell** (-diagramm) beschreibt, welche Klassen existieren und in welchen Beziehungen sie zueinander stehen.
- **Verschiedene Kontexte:**
  Fachklassen (aus der Analyse abgeleitet)
  ➔ **Designklassen** (Struktur des Lösungskonzeptes)
- Klassenmodelle werden durch Klassendiagramme visualisiert.

Klassendiagramme
Repräsentation von Klassen



- Klassen werden durch Rechtecke dargestellt, die entweder nur den Namen der Klasse (fettgedruckt, GroßGeschrieben, Camecasing) tragen
- oder zusätzlich:
- **`attribute`:** mindestens Namen, ggf. zusätzliche Elemente
- **`method()`**: mindestens Namen, ggf. zusätzliche Elemente

Zusätzliche Elemente bei Attributen (1)

- Für Attribute kann ein Datentyp (eine Klasse) angegeben werden
  Notation: `attribute_0 : Float`
  (in Python beschreibt dies, welche Klasse hier erwartet wird!)
- Attribute können **Initialwerte** haben.
  Notation: `attribute_1 = 0`
- Attribute können **Zusicherungen** haben.
  Hiermit können zusätzliche Beschränkungen der Wertemenge des Attributs ausgedrückt werden.
  Notation: werden separat notiert, z.B. als Kommentar

Zusätzliche Elemente bei Attributen (2)

- Attribute können **besondere Eigenschaftswerte** haben.
  Beispiel: `read only`
  [wäre in Python eine "Konstante":
  Namen nur in GROSSBUCHSTABEN]
- Optionale oder obligatorische Attribute können durch Angabe einer entsprechenden **Multiplizität** ausgezeichnet werden.

  Notation:
  ```
  optional_attribute: Class[0..1]
  mandatory_attribute: Class[1]
  ```

  Multiplizitätsangaben sollen nur notiert werden **wenn sie ungleich `[1]` sind**
  Eine Sortiereigenschaft kann mit `"unordered"` (default) oder `"ordered"` notiert werden.

Zusätzliche Elemente bei Attributen (3)

Attribute können **Sichtbarkeiten (Zugriffsrestriktionen)** haben.

Notation: erfolgt direkt vor dem Attributnamen

```
+ public_attribute
# protected_attribute
- private_attribute
~ package_attribute
```

"Wie realisieren Sie diese in Python?" ;-)

Klassenattribute werden unterstrichen

Notation: `class_attribute`

Beispiele für Attribute in UML

```
name : String = 'Unkonwn'
Born : Date
radius : Integer = 25 (readonly)
Default_name : 'Noame'
~ version_number : Integer
- counter : Integer
dynam_array[*] [ordered]
name : String[1]
first_name[0..1]
first_names[1..5]
```

Zusätzliche Elemente bei Methoden (1)

- Die Parameter einer Methode entsprechen in ihrer Definition den Attributen, d.h. sie tragen einen Namen und ggf. weitere Angaben , zum Beispiel zum Typ und Default-Werten.
- Die Parameter einer Methode können mit den **Parametern `in, out, oder inout`** gekennzeichnet werden.
  Notation:
  `get-position(out x : Integer, out y : Integer)`
- Ausgabewerte einer Methode (Rückgabewerte, Funktionsergebnisse) können wie Parameter ausgestattet sein:
  Notation:
  `set-position(out x : Float, out y : Float) : Boolean`

Zusätzliche Elemente bei Methoden (2)

- Methoden können mit **Zusicherungen** versehen werden, die beispielsweise beschreiben, welche Bedingungen beim Aufruf erfüllt sein müssen oder welche Werte die Parameter besitzen dürfen.
  Notation: Zusicherungen sind separat zu notieren.
- Methoden können mit **Eigenschaftswerten** versehen werden. Eigenschaftswerte sind z.B. `{deprecated}` um auszudrücken, dass diese Methode nur noch zur Kompatibilität mit früheren Versionen existiert aber sonst nicht mehr verwendet werden soll.
  Notation: Eigenschaftswerte stehen in geschweiften Klammern
  `anzeigen {deprecated}`

Zusätzliche Elemente bei Methoden (3)

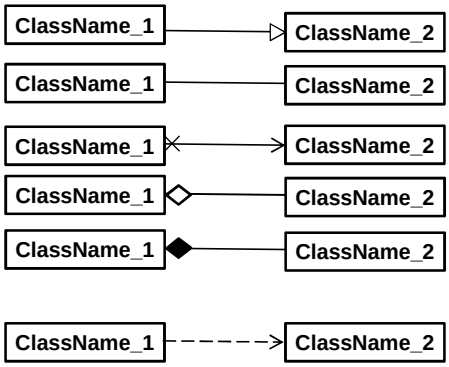
- Methoden besitzen **Sichtbarkeiten**.
  Notation: wie bei Attributen: + - ~ #
- Methoden besitzen **Multiplizitäten**.
  Notation: wie bei Attributen:

Beispiele für Methoden in UML

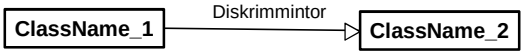
```
get_position()
get_position(out x : integer, out y : integer)
enlarge(factor : Float = 1.5): GeomFigure
+ add_phone_number(number : String)
# confer_right_to_use(): status_of_contract
```

Übersicht:
Beziehungselemente im Klassendiagramm

- **Vererbung**
- Assoziation
  - Gerichtete Assoziation
  - Aggregation
  - Komposition
- Abhängigkeit

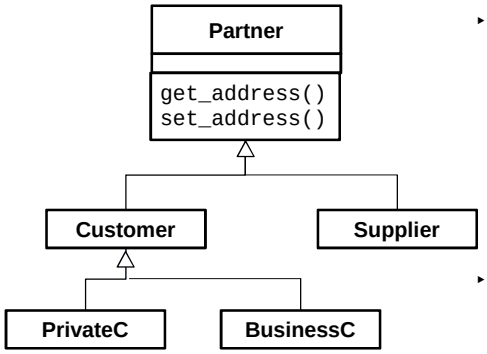


Vererbung



- Beschreibt eine Generalisierung resp. Spezialisierung
- ClassName_2 ist die Oberklasse (allgemeinere Eigenschaften)
- Es kann ein **Diskrimminator (*Generalisation Set Name*)** hinzugefügt werden, der den maßgeblichen Aspekt für die hierarchische Gliederung beschreibt.
- Anwendung ist zu empfehlen ➔ **dokumentierte Entwurfsentscheidung**

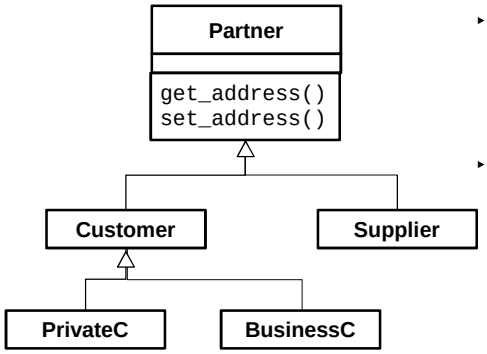
Klassen-Hierarchien



- Dieses Beispiel zeigt eine einfache Klassen-Hierarchie, bei der **Customer** und **Supplier** Erweiterungen von **Partner** sind. **Customer** ist wiederum eine Verallgemeinerung von **PrivateCustomer** und **BusinessCustomer**.
- Alle Erweiterungen erben die Methoden **getAddress()** und **setAddress()** von der Basis-Klasse.

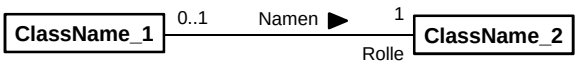
.

Klassen-Hierarchien

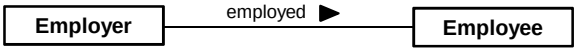


- Dieser Entwurf erlaubt es, Kontakt-Adressen verschiedener Sorten von Partnern z.B. in einer Liste zu verwalten.
- Damit bleibt z.B. der Ausdruck von Adressen unabhängig von den vorhandenen Ausprägungen.

Assoziation (*link*)

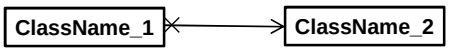


- beschreibt eine Verbindung (Beziehungen) zwischen Klassen
- kann auch rekursiv sein (auf sich selbst bezogen)
- Jede Assoziation kann einen Namen haben
  Bsp.



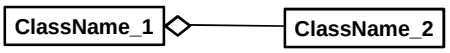
- Es können auch Rollen und Multiplizitätsangaben angegeben werden.
- Spezielle Varianten der Assoziation sind die Aggregation und die Komposition (siehe unten)

Gerichtete Assoziation



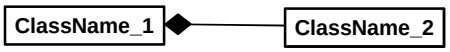
- **Bedeutung der Pfeilspitze:**
  hier kann von **ClassName_1** direkt zu **ClassName_2** navigiert werden.
- **Bedeutung des Kreuzes:**
  zu ClassName_1 kann von ClassName_2 **explizit** nicht navigiert werden.

Aggregation



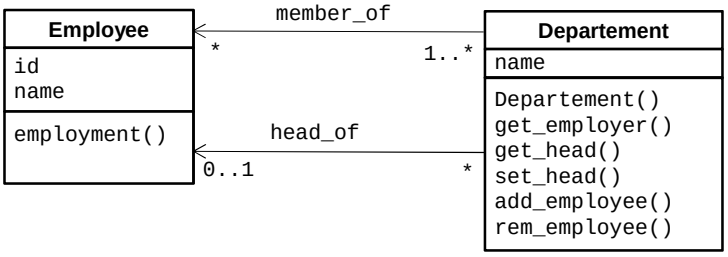
- Beschreibt eine **Ganzes-Teile-Hierarchie,**
  d.h. das Ganze ClassName_1 besteht aus ClassName_2 Teilen
- Raute steht auf der Seite des Aggregats
- Meist ist die Multiplizität auf Seiten des Ganzes = 1 und
  wird i.d.R. dann weggelassen

Komposition



- Eine Komposition ist eine spezielle Aggregation, bei der die Teile vom Ganzen existenzabhängig sind.
- Die Kardinalität (oder Multiplizität) auf Seite des Ganzen muss immer 0 oder 1 sein. Jedes Teil ist nur Teil maximal eines Kompositionsobjektes.
- Beispiel: Rechnung mit den Rechnungspositionen
- Die Rechnungspositionen sind existentsabhängig von der Rechnung. Sobald die Rechnung gelöscht wird, werden auch alle Rechnungspositionen gelöscht.

Beispiel: Klassen-Diagramme



Klassen-Diagramme bestehen aus Klassen (dargestellt als Rechtecke) und deren Beziehungen (Linien und Pfeile) untereinander.

Hinweis

- Man kann sehr viele Details in Klassendiagrammen unterbringen.
- Bei größeren Projekten sollte **nicht** der Versuch unternommen werden, alle Details in ein großes Diagramm zu integrieren.
- Stattdessen ist es sinnvoller, zwei oder mehr Ebenen von Klassen-Diagrammen zu haben, die sich entweder auf die Übersicht oder die Details in einem eingeschränkten Bereich konzentrieren.

Beispiel: Darstellung einer Klasse

Departement
name
Departement() get_employer() get_head() set_head() add_employee() rem_employee()

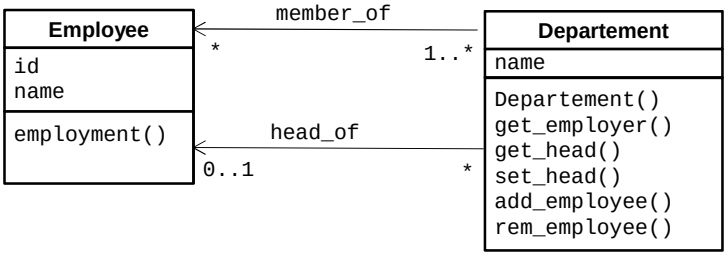
- Die Rechtecke für eine Klasse spezifizieren den
  - Namen der Klasse und die
  - öffentlichen) Attribute und
  - öffentlichen) Methoden.
- Die erste Methode sollte (sofern vorhanden) der Konstruktor sein.
- Diese Sektionen werden durch horizontale Striche getrennt.

Beispiel: Darstellung einer Klasse

Departement
name
Departement() get_employer() get_head() set_head() add_employee() rem_employee()

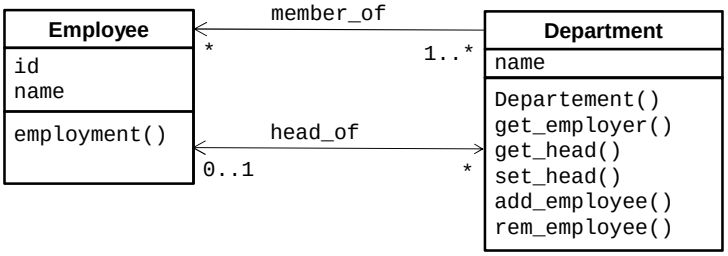
- Bei einem **Übersichtsdiagramm** ist es auch üblich, nur den Klassennamen anzugeben.
- Private Felder und private Methoden werden normalerweise weggelassen.
- Eine Ausnahme ist nur angemessen, wenn eine Dokumentation für das Innenleben einer Klasse angefertigt wird, wobei dann auch nur das Innenleben einer einzigen Klasse gezeigt werden sollte.

Beispiel: Darstellung einer Klasse



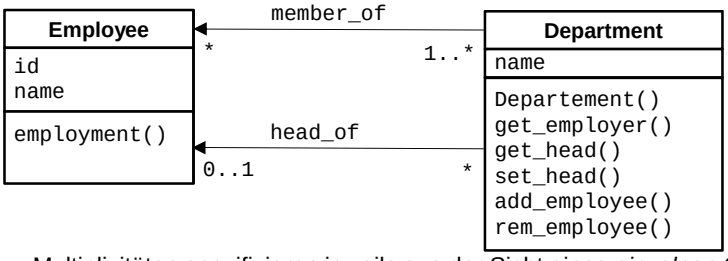
- Primär werden bei den dargestellten Beziehungen Referenzen in der Datenstruktur berücksichtigt.

Darstellung einer Klasse



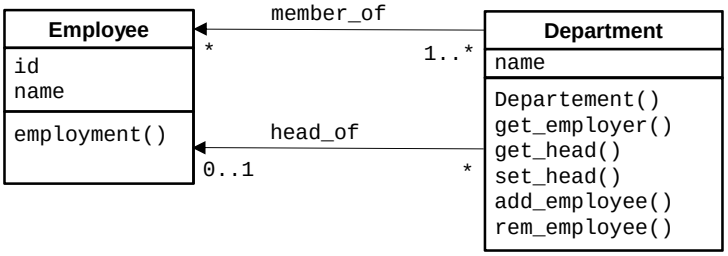
- Referenzen werden mit durchgezogenen Linien dargestellt, wobei diese die Verweisrichtung angeben.
- In diesem Beispiel kann ein Objekt der Klasse **Department** eine Liste von zugehörigen Angestellten und den Abteilungsleiter liefern.

Komplexitätsgrade



- Multiplizitäten spezifizieren jeweils aus der Sicht eines *einzelnen* Objekts, wie viele konkrete Beziehungen zu Objekten der anderen Klasse existieren können.
- Eine Multiplizität wird in Form eines Intervalls angegeben (z.B. "0..1"), in Form einer einzelnen Zahl oder mit "*" als Kurzform für beliebig.

Klassendiagramme



- Für jede Beziehung werden zwei Komplexitätsgrade angegeben, jeweils aus Sicht eines Objekts der beiden beteiligten Klassen.
- In diesem Beispiel hat eine Abteilung gar keinen oder einen Leiter, aber ein Angestellter kann für beliebig viele Abteilungen die Rolle des Leiters übernehmen.

Implementierung von Komplexitätsgraden

- Bei der Implementierung ist der Komplexitätsgrad am Pfeilende relevant:
- Ein Komplexitätsgrad von 1 wird typischerweise durch eine private Referenz, die auf ein Objekt der anderen Klasse zeigt, repräsentiert. Dieser Zeiger muss dann immer wohldefiniert sein und auf ein Objekt zeigen.
- Bei einem Grad von 0 oder 1 darf der "Zeiger" auch **NIL** (oder **NULL**) sein.

Weitere Anmerkungen

- Bei ***** werden Listen oder andere geeignete Datenstrukturen benötigt, um alle Verweise zu verwalten. Solange für die Listen vorhandene Sprachmittel oder Standard-Bibliotheken für Container verwendet werden, werden sie selbst nicht in das Klassendiagramm aufgenommen.
- Im Beispiel hat die Klasse **Department** einen privaten Zeiger **head**, der entweder **NIL** ist oder auf einen **Employee** zeigt.
- Für die Beziehung **memberOf** wird hingegen bei der Klasse **Department** eine Liste benötigt

Konsistenz bei Komplexitätsgraden

- Auch der Komplexitätsgrad am Anfang des Pfeiles ist relevant, da er angibt, wie viel Verweise insgesamt von Objekten der einen Klasse auf ein einzelnes Objekt der anderen Klasse auftreten können.
- Im Beispiel muss jeder Angestellte in mindestens einer Abteilung aufgeführt sein. Er darf aber auch in mehreren Abteilungen beheimatet sein.
- Um die Konsistenz zu bewahren, darf der letzte Verweis einer Abteilung zu einem Angestellten nicht ohne weiteres gelöscht werden. Dies ist nur zulässig, wenn auch gleichzeitig der Angestellte gelöscht wird oder in eine andere Abteilung aufgenommen wird.
- **Die Klasse, von der ein Pfeil ausgeht, ist üblicherweise für die Einhaltung der zugehörigen Komplexitätsgrade verantwortlich.**

Zwischen-Zusammenfassung

- Die Objektorientierung bedeutet eine Modularisierung der Programme und eine klare Beschreibung ihrer Zusammenarbeit.
- Bei der Planung einer Software helfen Anwendungsszenarien (Use Cases) weiter. Sie beschreiben Akteure und ihre Interaktionen.
- Die Beziehungen zwischen Objekten können mit UML-Klassendiagrammen verdeutlicht werden.
- … am Montag kommt noch etwas mehr!