- Mit der Einführung verschiedener objekt-orientierter Programmiersprachen entstanden auch mehr oder weniger formale graphische Sprachen für OO-Designs.
- Popularität genossen unter anderem die graphische Notation von Grady Booch aus dem Buch *Object-Oriented Analysis and Design*, OMT von James Rumbaugh (Object Modeling Technique), die Diagramme von Bertrand Meyer in seinen Büchern und die Notation von Wirfs-Brock et al in *Designing Object-Oriented Software*.
- Später vereinigten sich Grady Booch, James Rumbaugh und Ivar Jacobson in ihren Bemühungen, eine einheitliche Notation zu entwerfen. Damit begann die Entwicklung von UML Mitte der 90er Jahre.

- UML wird als Standard von der Object Management Group (OMG) verwaltet.
- Die Version 2.5.1 ist die aktuelle Fassung von Dezember 2017.
- Zu dem Standard gehören mehrere Dokumente, wovon für uns insbesondere die *OMG UML Superstructure* interessant ist: Im Abschnitt 9 werden Klassendiagramme beschrieben, im Abschnitt 17.8 Sequenzdiagramme und im Abschnitt 18 Use Cases.
- Bei den Abschnitten werden jeweils im Unterabschnitt 3 die einzelnen Elemente eines Diagramms beschrieben und im Unterabschnitt 4 die einzelnen graphischen Elemente einer Diagrammart tabellarisch zusammengefasst.
- Die einzelnen Dokumente des Standards lassen sich von `http://www.omg.org/` herunterladen.

- 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 beschränken uns im Rahmen der Vorlesung auf nur drei Diagrammarten, die eine größere Verbreitung erfahren haben und dort jeweils nur auf eine kleine Teilmenge der Ausdrucksmöglichkeiten.

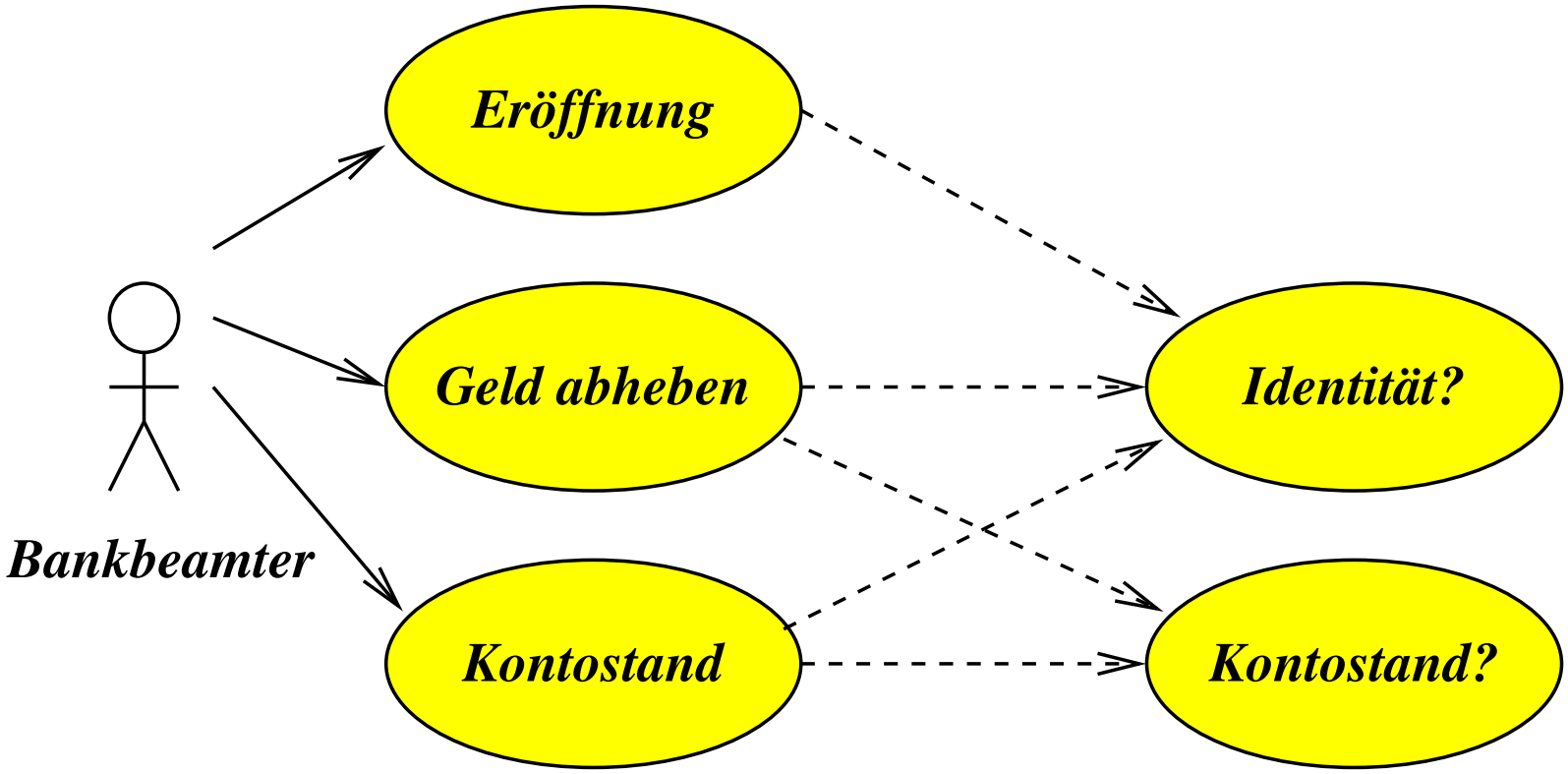
- *Use Cases* dokumentieren während der Analyse die typischen Prozeduren aus der Sicht der aktiven Teilnehmer (Akteure) für ausgewählte Fälle.
- Akteure sind aktive Teilnehmer, die Prozesse in Gang setzen oder Prozesse am Laufen halten.

- Akteure können
  - von Menschen übernehmbare Rollen, die direkt interaktiv mit dem System arbeiten,
  - andere Systeme, die über Netzwerkverbindungen kommunizieren, oder
  - interne Komponenten sein, 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.

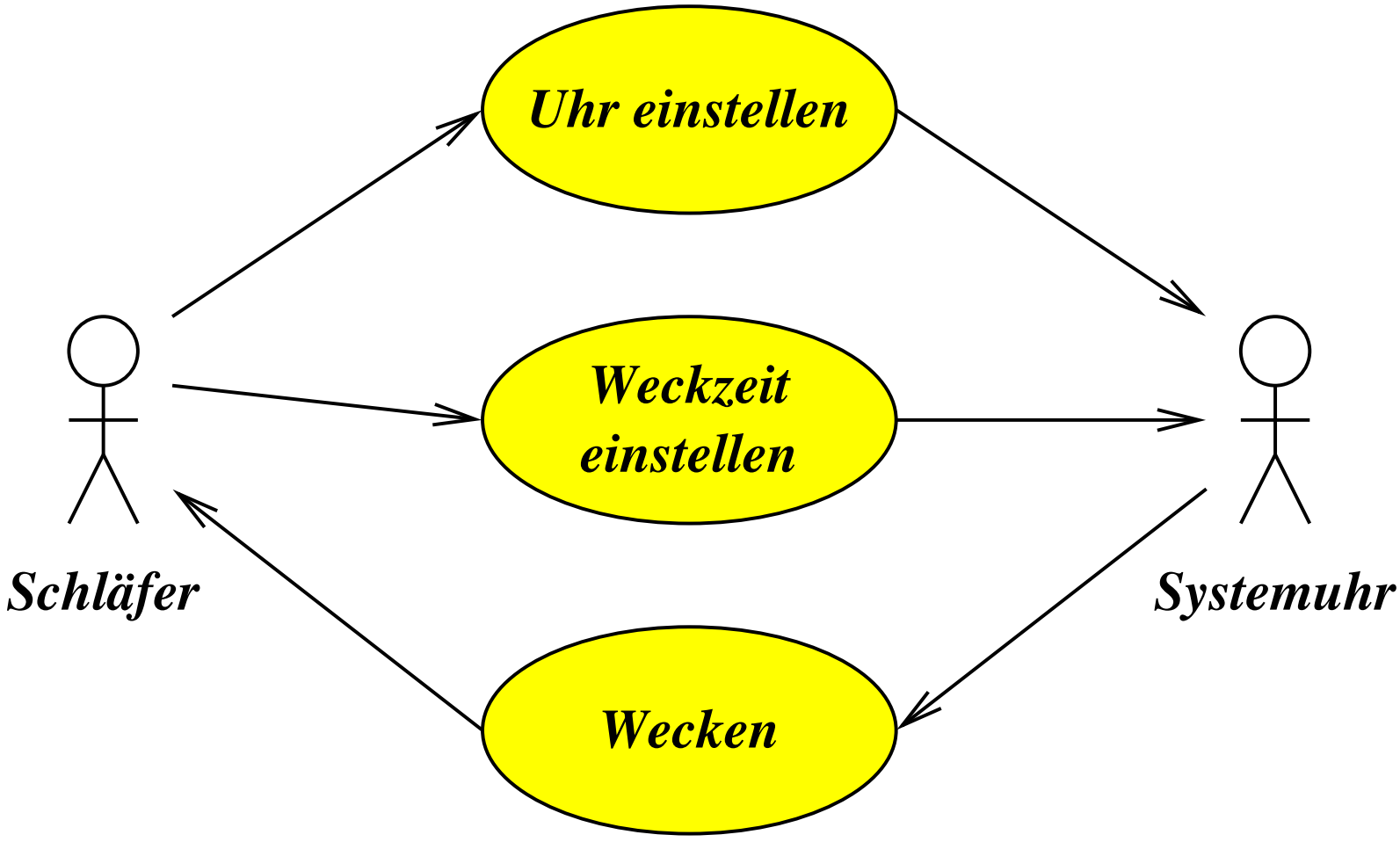
Aus welchen für die Nutzer sichtbaren Schritten bestehen einzelne typische Abläufe bei dem Umgang mit Bankkunden?

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

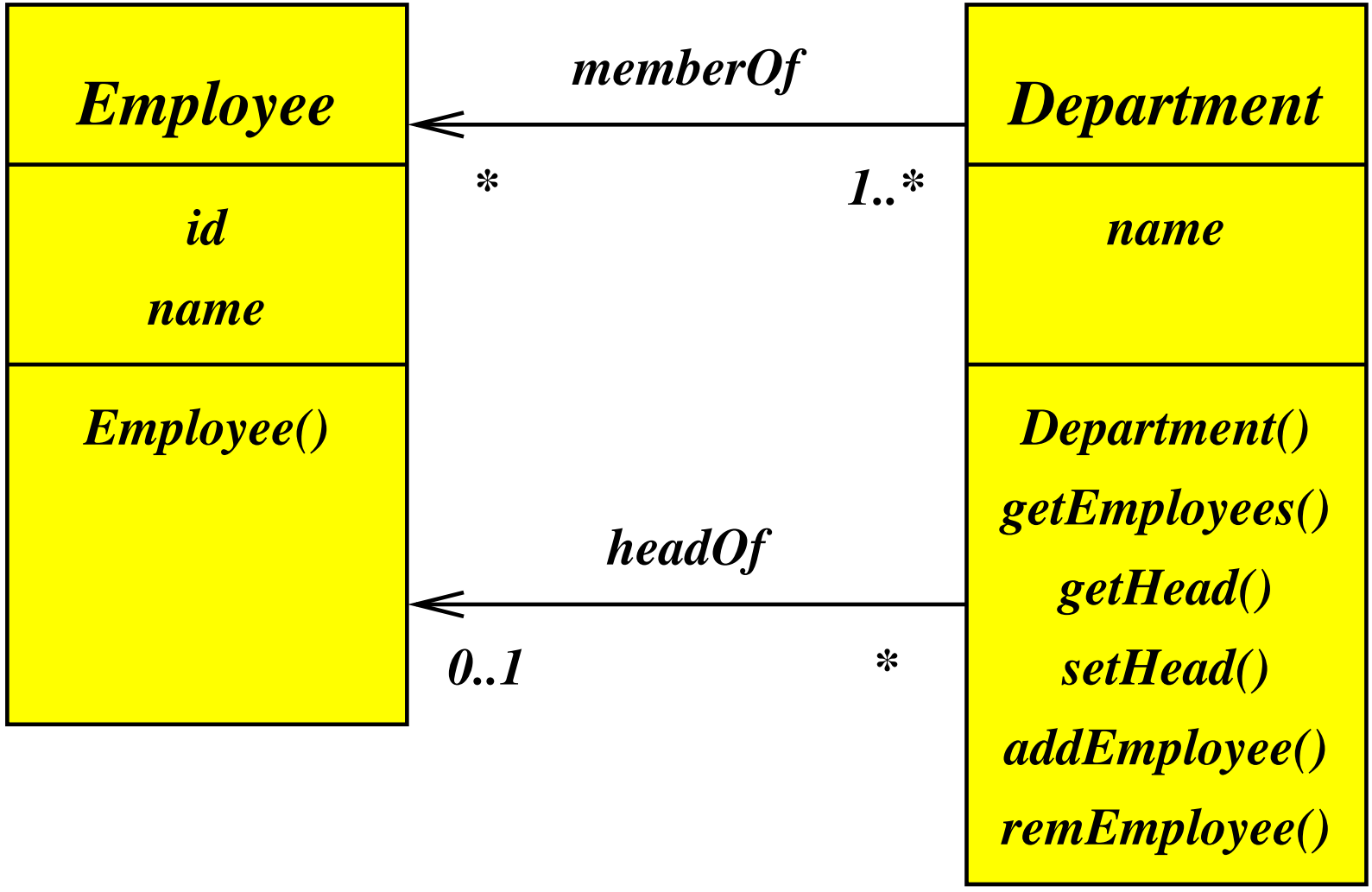
- Hier wurden nur die Aktivitäten aufgeführt, die der Schalterbeamte im Umgang mit dem System ausübt.
- Der Akteur ist hier der Schalterbeamte, weil er in diesen Fällen mit dem System arbeitet. Der Kunde wird nur dann zum Akteur, wenn er beispielsweise am Bankautomaten steht oder über das Internet auf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


Eröffnung
Geld abheben
Kontostand
Identität?
Kontostand?
Bankbeamter

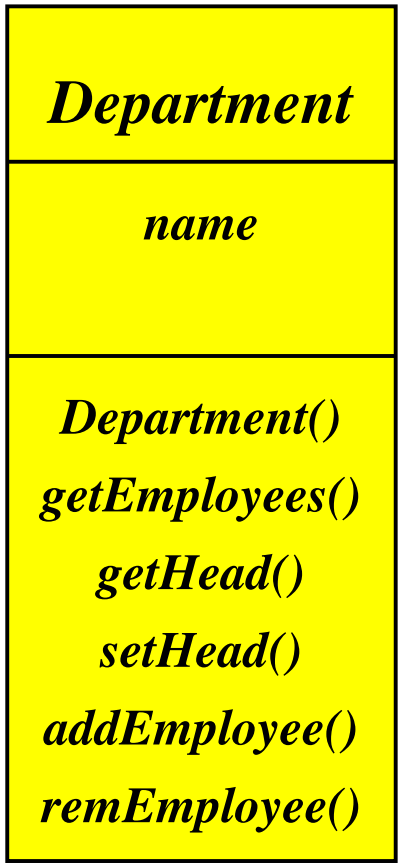
- Eine glatte Linie mit einem Pfeil verbindet einen Akteur mit einem Use-Case. Das bedeutet, dass die mit dem Use-Case verbundene Prozedur von diesem Akteur angestoßen bzw. durchgeführt wird.
- Gestrichelte Linien repräsentieren Beziehungen zwischen mehreren Prozeduren. Damit können Gemeinsamkeiten hervorgehoben werden.
- Wichtig: Pfeile repräsentieren **keine** Flußrichtungen von Daten. Es führt hier insbesondere kein Pfeil zu dem Bankbeamten zurück.
- Bei neueren UML-Versionen fallen die Pfeile weg, weil sie letztlich redundant sind.


Uhr einstellen
Weckzeit einstellen
Wecken
Schläfer
Systemuhr

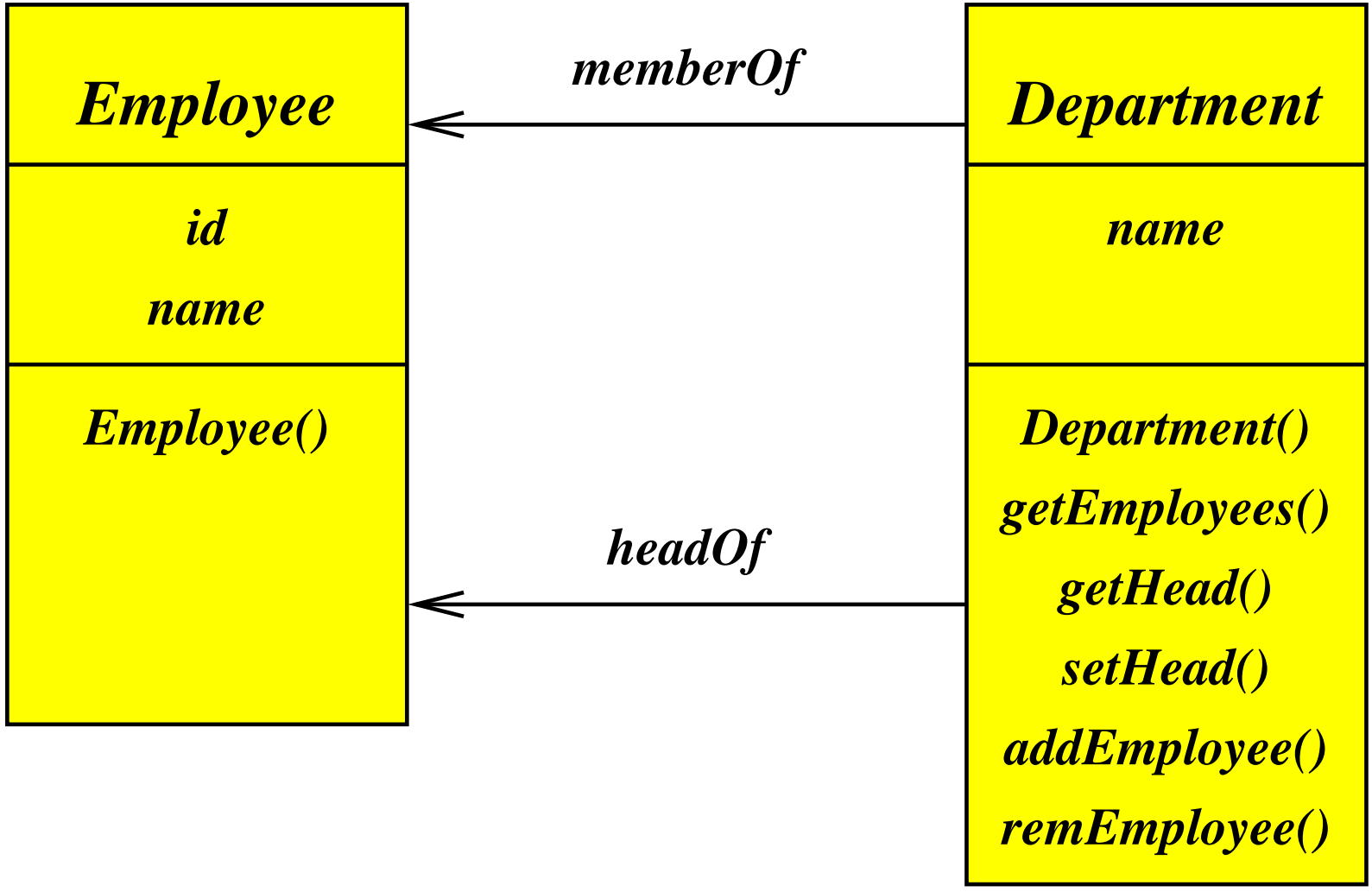
- Es können auch Pfeile von Prozeduren zu Akteuren gehen, wenn sie eine Benachrichtigung repräsentieren, die sofort wahrgenommen wird.
- Ein Wecker hat intern einen Akteur — die Systemuhr. Sie aktualisiert laufend die Zeit und muß natürlich eine Neu-Einstellung der Zeit sofort erfahren.
- Das Auslösen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


Employee
id
name
Employee()
memberOf
*
1..*
Department
name
Department()
getEmployees()
getHead()
setHead()
addEmployee()
remEmployee()
headOf
0..1
*

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

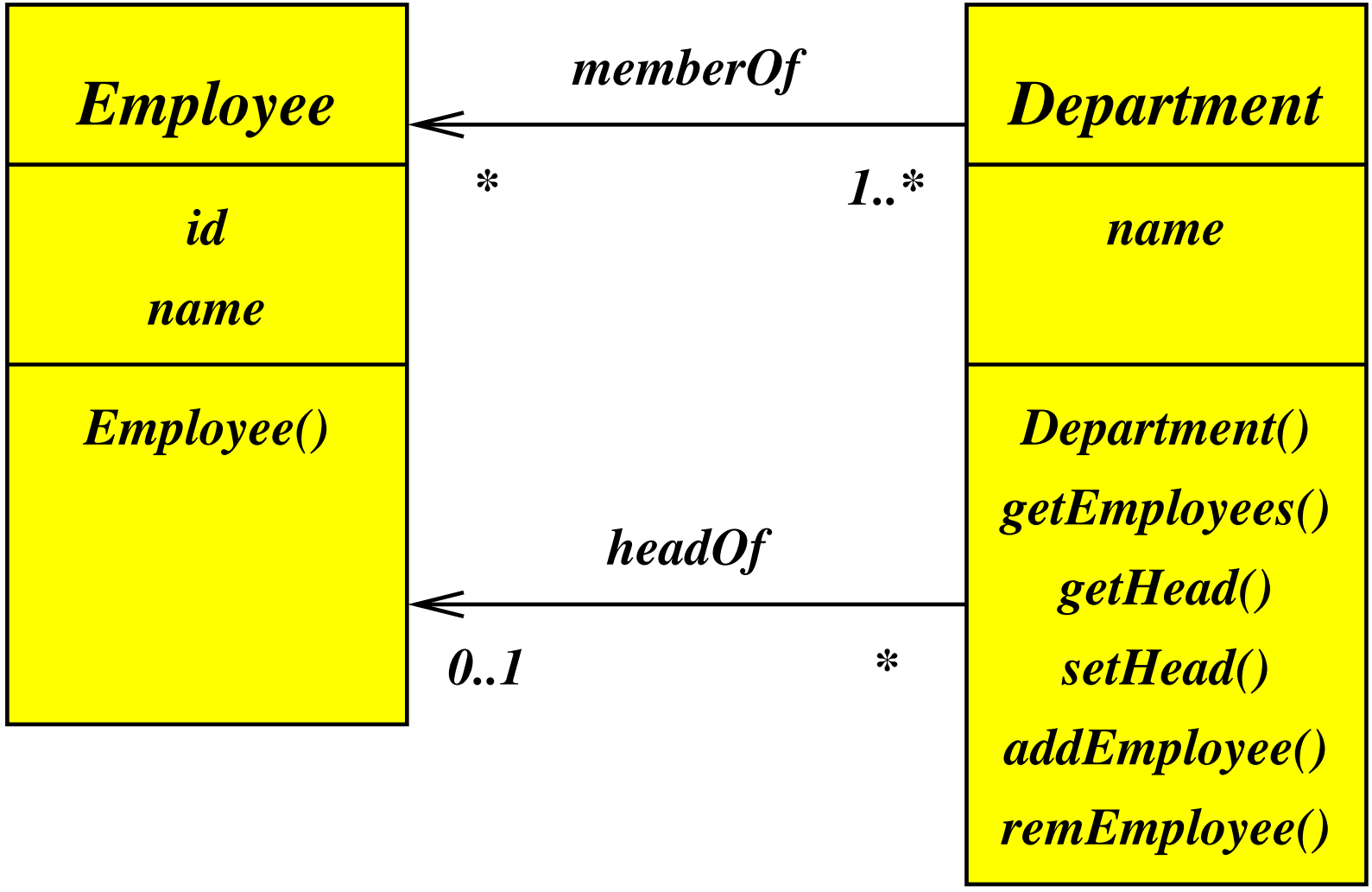


- Die Rechtecke für eine Klasse spezifizieren den Namen der Klasse und die öffentlichen Felder und Methoden. Die erste Methode sollte (sofern vorhanden) der Konstruktor sein.
- Diese Sektionen werden durch horizontale Striche getrennt.
- Bei einem Übersichtsdiagramm ist es auch üblich, nur den Klassennamen 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.


Employee
id
name
Employee()
memberOf
headOf
Department
name
Department()
getEmployees()
getHead()
setHead()
addEmployee()
remEmployee()

- Primär werden bei den dargestellten Beziehungen Referenzen in der Datenstruktur berücksichtigt.
- Referenzen werden mit durchgezogenen Linien dargestellt, wobei ein oder zwei Pfeile die Verweisrichtung angeben.
- In diesem Beispiel kann ein Objekt der Klasse *Department* eine Liste von zugehörigen Angestellten liefern.
- Zusätzlich ist es mit gestrichelten Linien möglich, die Benutzung einer anderen Klasse zum Ausdruck zu bringen. Ein typisches Beispiel ist die Verwendung einer fremden Klasse als Typ in einer Signatur.
- Beziehungen reflektieren Verantwortlichkeiten. Im Beispiel ist die Klasse *Department* für die Beziehungen *memberOf* und *headOf* zuständig, weil die Pfeile von ihr ausgehen.
- Eine Beziehung kann beidseitig mit Pfeilen versehen sein, dann sind beide Klassen dafür verantwortlich.

- Durch die Beziehungen wird typischerweise sichtbar, wie eine Navigation durch eine Datenstruktur möglich ist. Es lässt sich somit die Frage beantworten, ob beginnend von einem Objekt einer Klasse eine Traverse über weitere Objekte möglich ist auf Basis der vorhandenen Beziehungen.
- Pfeilrichtungen werden in diesem Sinne gerne als potentielle Navigationsrichtungen interpretiert, d.h. die Klasse, von der aus ein Pfeil zu einer anderen Klasse ausgeht, sollte auch Methoden anbieten, mit der eine entsprechende Abfrage oder Traverse möglich ist.
- Eine Beziehung ohne Pfeile sagt nichts zu Verantwortlichkeiten oder Navigierbarkeit aus.
- Wenn Pfeile verwendet werden, sollten diese vollständig sein. Es erscheint wenig sinnvoll, nur die eine Richtung anzugeben, wenn sich eine Beziehung auch in der anderen Richtung navigieren lässt. Der UML-Standard lässt dies zu und verwendet stattdessen ein „x“ als Markierung für Nicht-Navigierbarkeit. Wir verzichten darauf.

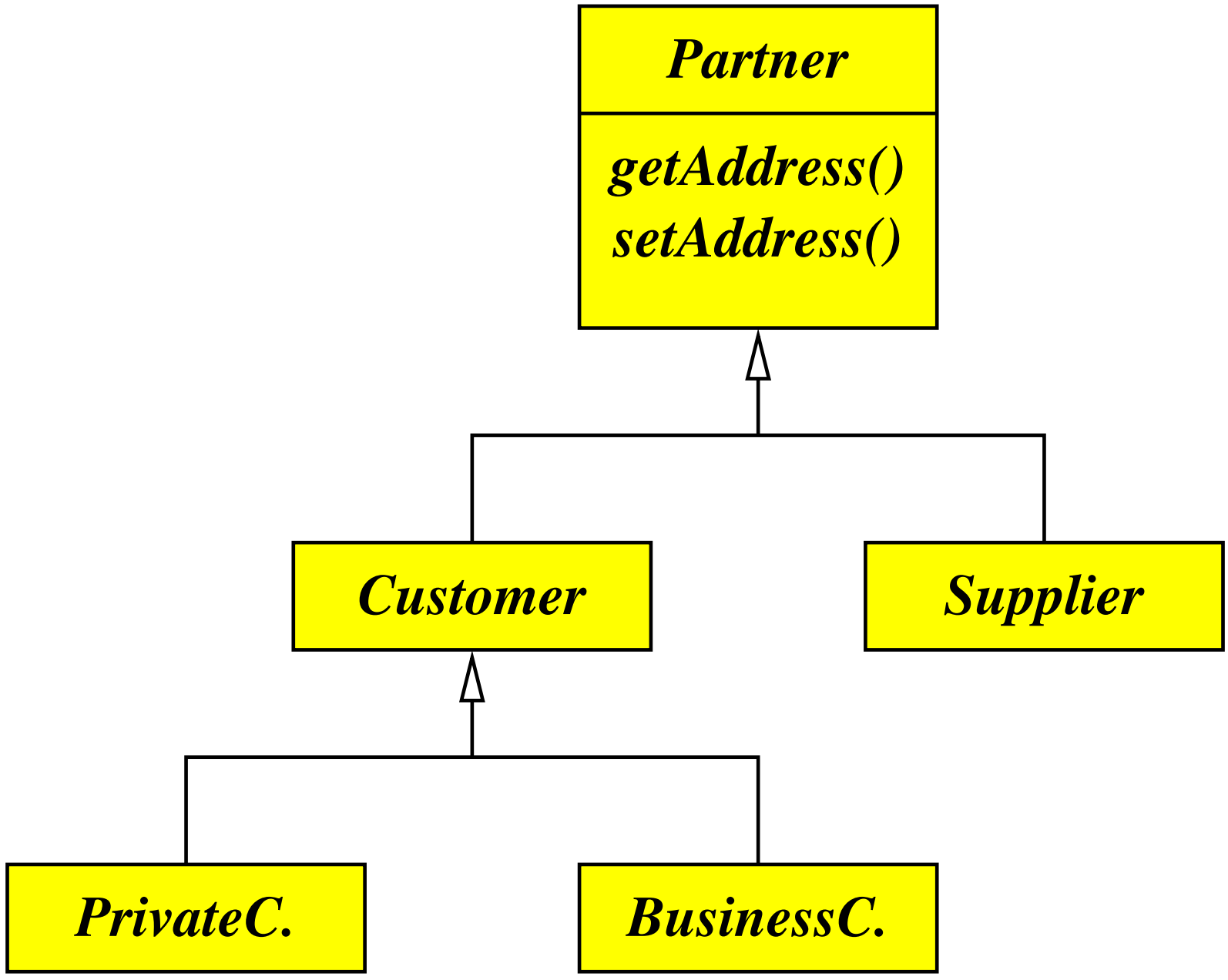

Employee
id
name
Employee()
memberOf
*
1..*
Department
name
Department()
getEmployees()
getHead()
setHead()
addEmployee()
remEmployee()
headOf
0..1
*

- Komplexitätsgrade spezifizieren jeweils aus der Sicht eines *einzelnen* Objekts, wieviele konkrete Beziehungen zu Objekten der anderen Klasse existieren können.
- Ein Komplexitätsgrad wird in Form eines Intervalls angegeben (z.B. „0..1“), in Form einer einzelnen Zahl oder mit „*“ als Kurzform für 0 bis unendlich.
- 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.

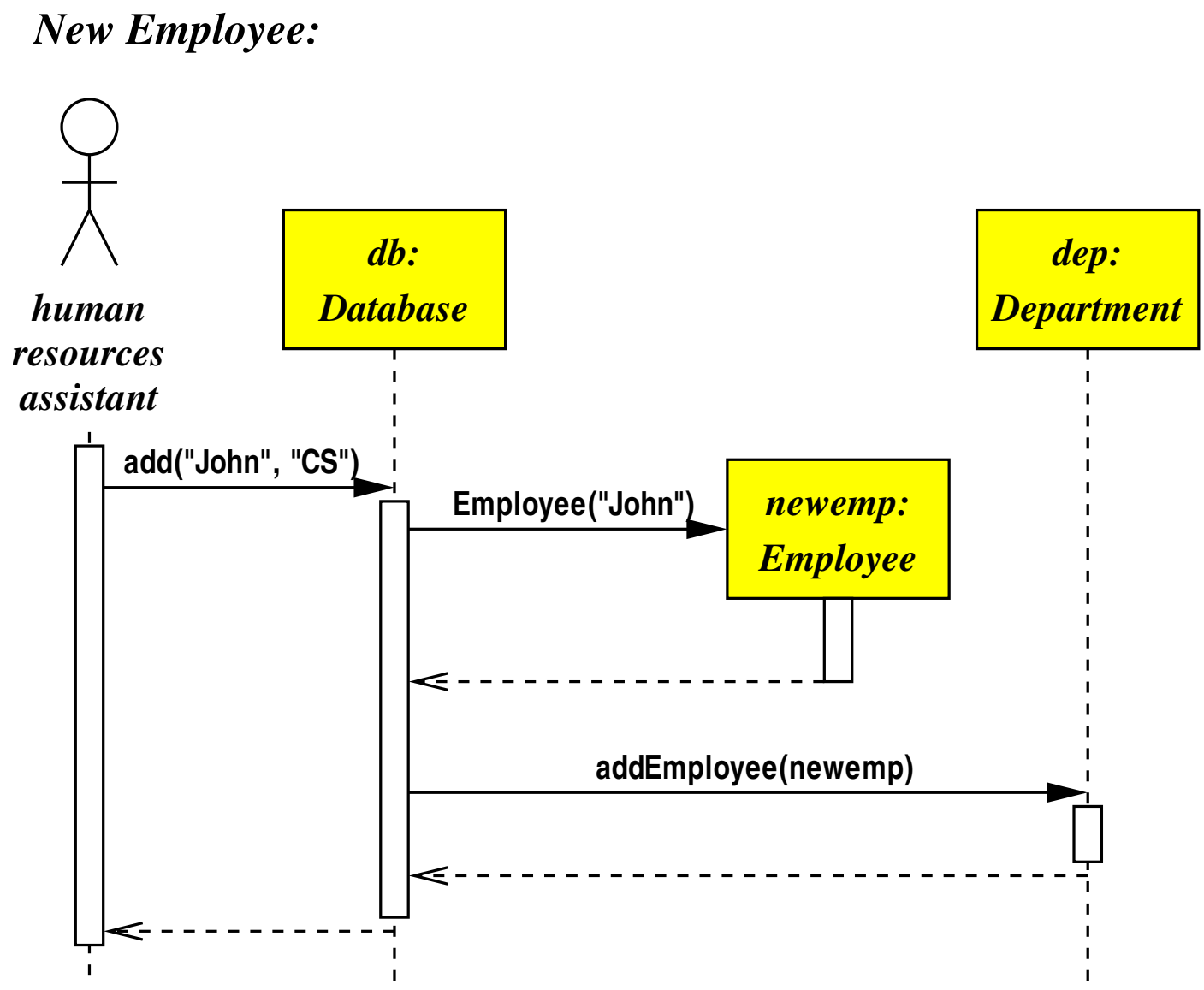
Bei der Implementierung ist der Komplexitätsgrad am Pfeilende relevant:

- 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 **nullptr** sein.
- 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 **nullptr** ist oder auf einen *Employee* zeigt.
- Für die Beziehung *memberOf* wird hingegen bei der Klasse *Department* eine Liste benötigt.

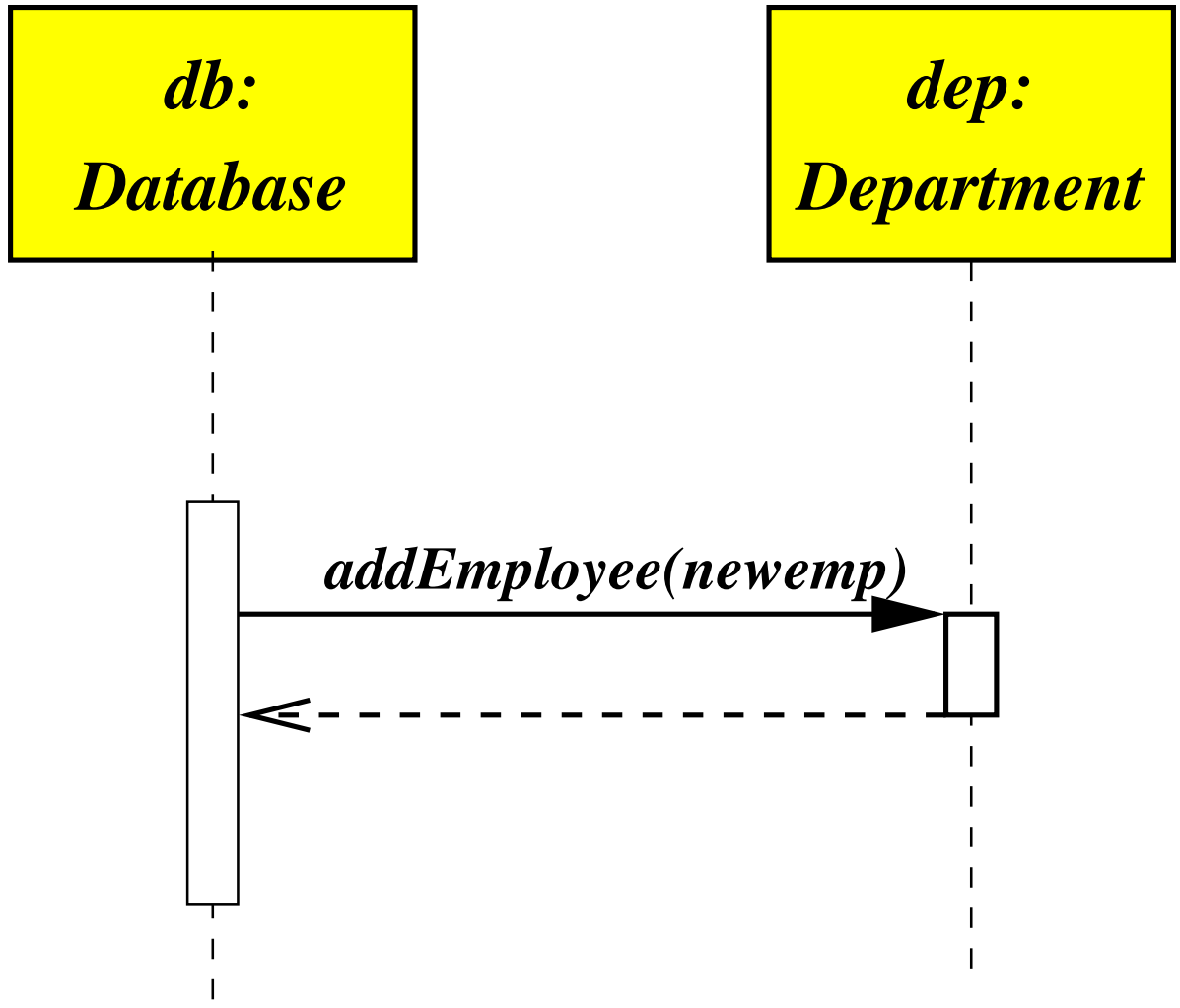
- Auch der Komplexitätsgrad am Anfang des Pfeiles ist relevant, da er angibt, wieviel Verweise insgesamt von Objekten der einen Klasse auf ein einzelnes Objekt der anderen Klasse auftreten können.
- Im Beispiel muß jeder Angestellte in mindestens einer Abteilung aufgeführt sein. Er darf aber auch in 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.


Partner
getAddress()
setAddress()
Customer
Supplier
PrivateC.
BusinessC.

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

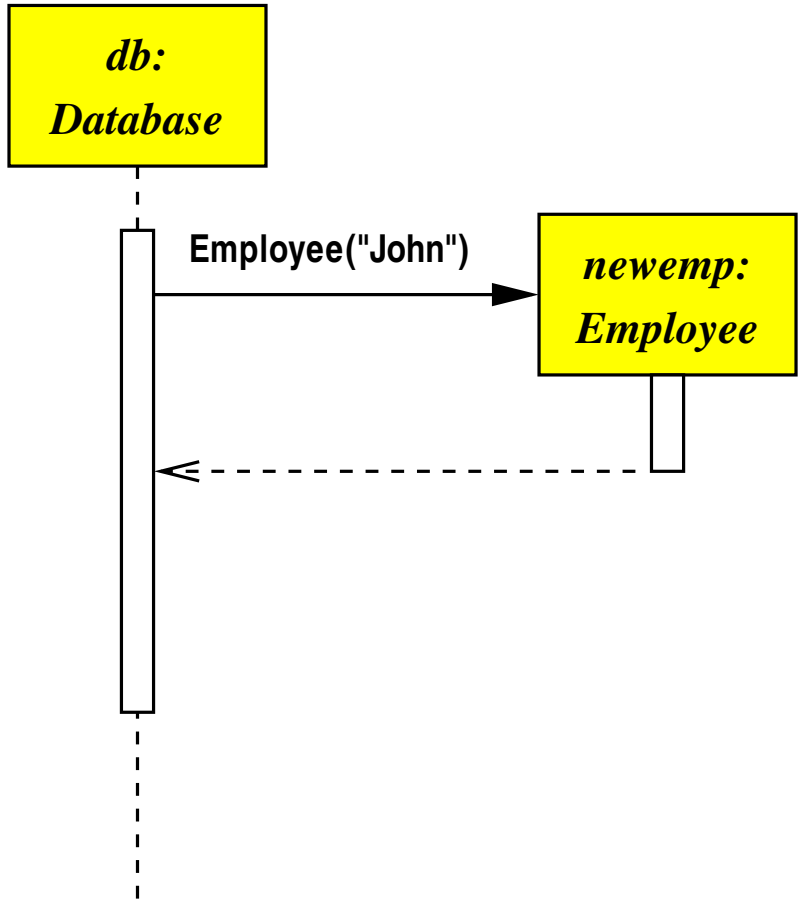

New Employee:
human resources assistant
db: Database
dep: Department
add("John", "CS")
Employee("John")
newemp: Employee
addEmployee(newemp)

- Sequenz-Diagramme zeigen den Kontrolfluss für ausgewählte Szenarien.
- Die Szenarien können unter anderem von den Use-Cases abgeleitet werden.
- Sie demonstrieren, wie Akteure und Klassen miteinander in einer sequentiellen Form operieren.
- Insbesondere wird die zeitliche Abfolge von Methodenaufrufen für einen konkreten Fall dokumentiert.
- Sequenz-Diagramme helfen dabei zu ermitteln, welche Methoden bei den einzelnen Klassen benötigt werden, um eine Funktionalität umzusetzen.


db:
Database
dep:
Department
addEmployee(newemp)

- Die Zeitachse verläuft von oben nach unten.
- Jedes an einem Szenario beteiligte Objekt wird durch ein Rechteck dargestellt, das die Klassenbezeichnung und optional einen Variablennamen enthält.
- Die Zeiträume, zu denen ein Objekt nicht aktiv ist, werden mit einer gestrichelten Linie dargestellt.
- Ein Objekt wird dann durch einen Methodenaufruf aktiv. Der Zeitraum, zu dem sich eine Methode auf dem Stack befindet, wird durch ein langgezogenes ein Rechteck dargestellt.
- Der Methodenaufruf selbst wird durch einen Pfeil dargestellt, der mit dem Aufruf beschriftet wird.
- Die Rückkehr kann entweder weggelassen werden oder sollte durch eine gestrichelte Linie markiert werden.

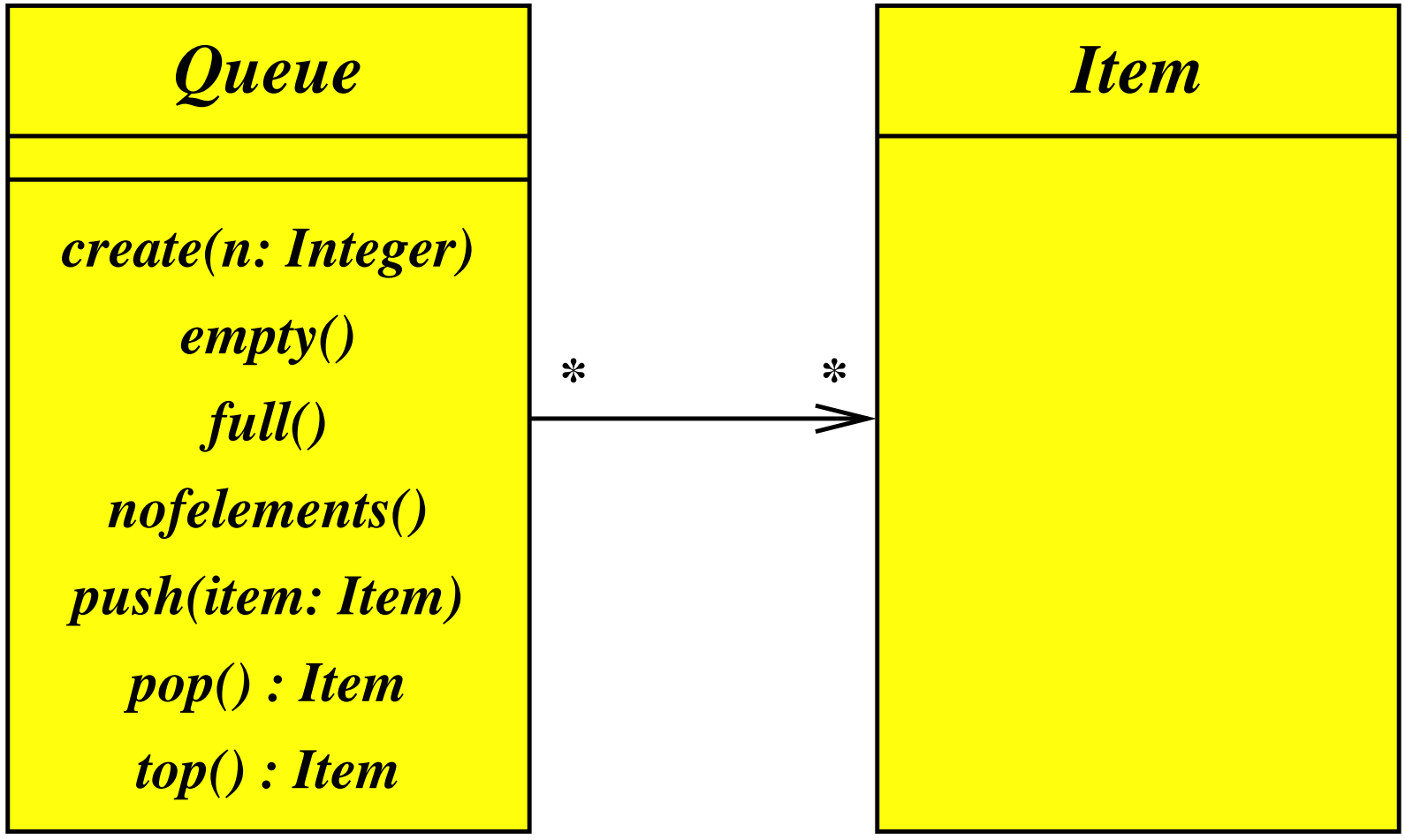
Konstruktoren in einem Sequenz-Diagramm



- Objekte, die erst im Laufe des Szenarios durch einen Konstruktor erzeugt werden, werden rechts neben dem Pfeil plaziert.
- Ganz oben stehen nur die Objekte, die zu Beginn des Szenarios bereits existieren.
- Da ein neu erzeugtes Objekt sofort aktiv ist, gibt es keine gestrichelte Linie zwischen dem Objekt und der ersten durch ein weißes Rechteck dargestellten Aktivitätsphase.

- Der Begriff des Vertrags (*contract*) in Verbindung von Klassen wurde von Bertrand Meyer in seinem Buch *Object-oriented Software Construction* und in seinen vorangegangenen Artikeln geprägt.
- Die Idee selbst basiert auf frühere Arbeiten über die Korrektheit von Programmen von Floyd, Hoare und Dijkstra.
- Wenn wir die Schnittstelle einer Klasse betrachten, haben wir zwei Parteien, die einen Vertrag miteinander abschließen:
  - Die Klienten, die die Schnittstelle nutzen, und
  - die Implementierung selbst mitsamt all den Implementierungen der davon abgeleiteten Klassen.

- Dieser Vertrag sollte explizit in formaler Weise im Rahmen der Schnittstellengestaltung einer Klasse spezifiziert werden. Er besteht aus:
  - Vorbedingungen (*preconditions*), die spezifizieren, welche Vorraussetzungen zu erfüllen sind, bevor eine Methode aufgerufen werden darf.
  - Nachbedingungen (*postconditions*), die spezifizieren, welche Bedingungen nach dem Aufruf der Methode erfüllt sein müssen.
  - Klasseninvarianten, die Bedingungen spezifizieren, die von allen Methoden jederzeit aufrecht zu halten sind.


Queue
create(n: Integer)
empty()
full()
nofelements()
push(item: Item)
pop() : Item
top() : Item
*
*
Item

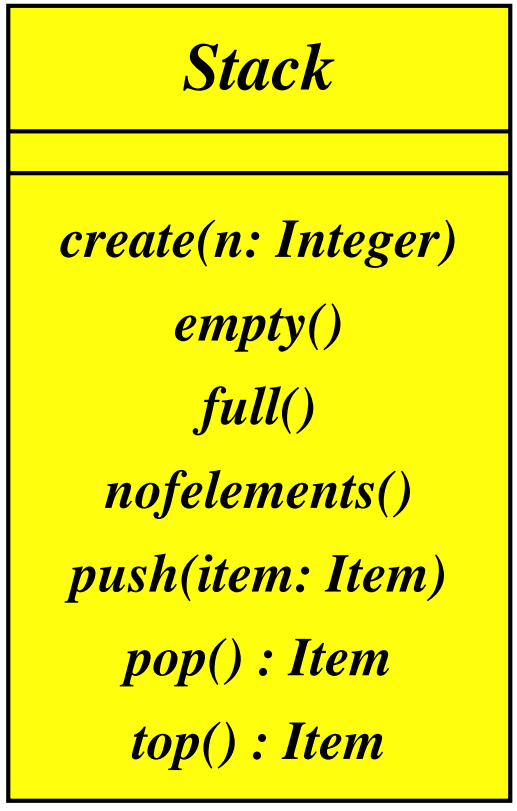
Methode	Vorbedingung	Nachbedingung
create()	$n > 0$	*empty*()&& *nofelements*()== 0
push()	!*full*()	!*empty*()&& *nofelements*() erhöht sich um 1
pop()	!*empty*()	*nofelements*() verringert sich um 1; beim i-ten Aufruf ist das i-te Objekt, das *push*() übergeben worden ist, zurückzuliefern.
top()	!*empty*()	*nofelements*() bleibt unverändert; liefert das Objekt, das auch bei einem nachfolgenden Aufruf von *pop*() geliefert werden würde

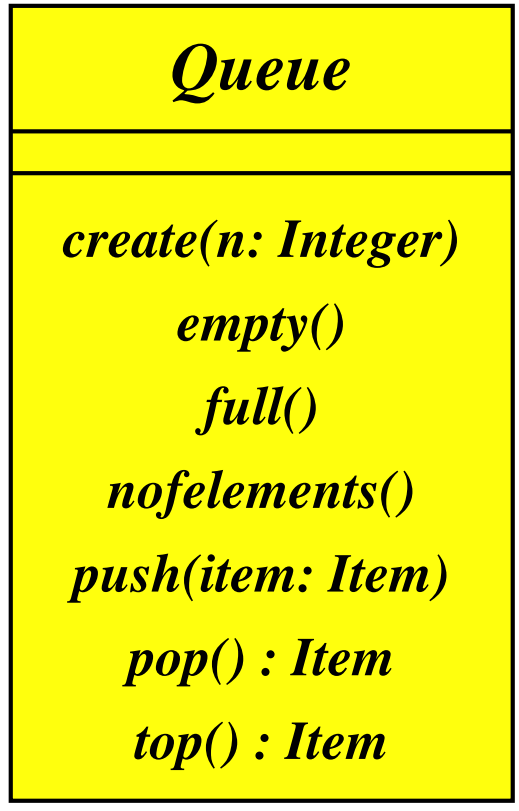
Klassen-Invarianten:

- *nofelements*()== 0 && *empty*()|| *nofelements*()> 0 && !*empty*()
- *empty*()&& !*full*()|| *full*()&& !*empty*()|| !*full*()&& !*empty*()
- *nofelements*()>= *n* || !*full*()

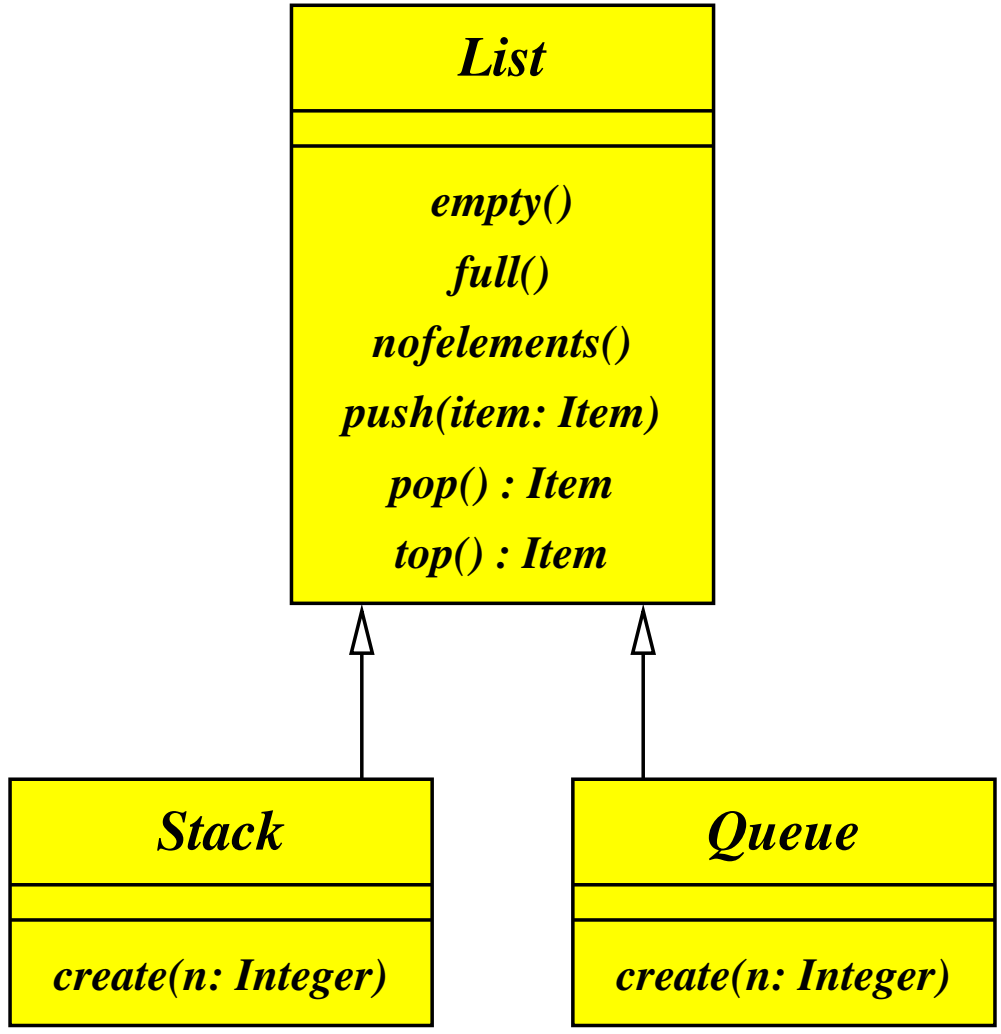
```
void Queue::push(const Item& item) {
   // precondition
   assert(!full());
   // prepare to check postcondition
   int before = nofelements();

   // ... adding item to the queue ...

   // checking postcondition
   int after = nofelements();
   assert(!empty() && after == before + 1);
}
```

- Teile des Vertrags können in Assertions verwandelt werden, die in die Implementierung einer Klasse aufzunehmen sind.
- Dies erleichtert das Finden von Fehlern, die aufgrund von Vertragsverletzungen entstehen.
- Der Verlust an Laufzeiteffizienz durch Assertions ist vernachlässigbar, solange diese nicht im übertriebenen Maße eingesetzt werden. (Das liegt u.a. auch daran, dass die Überprüfungen häufig parallelisiert ausgeführt werden können dank der Pipelining-Architektur moderner Prozessoren.)





- Signaturen alleine spezifizieren noch keine Klasse.
- Die gleiche Signatur kann mit verschiedenen Semantiken und entsprechend unterschiedlichen Verträgen assoziiert werden.

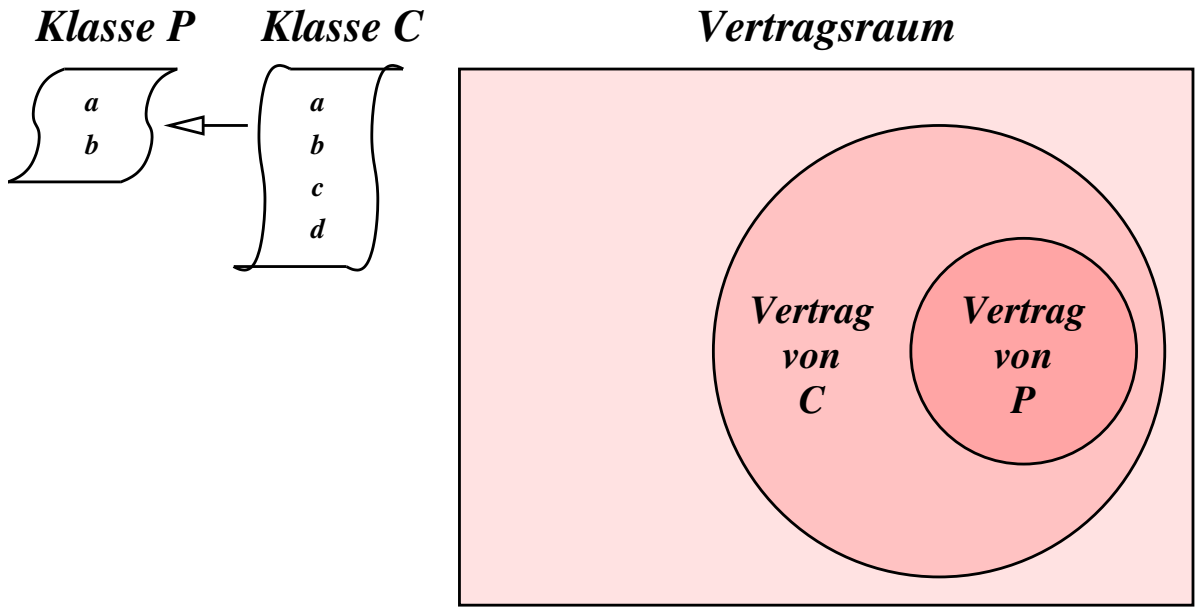

List
empty()
full()
nofelements()
push(item: Item)
pop() : Item
top() : Item
Stack
create(n: Integer)
Queue
create(n: Integer)

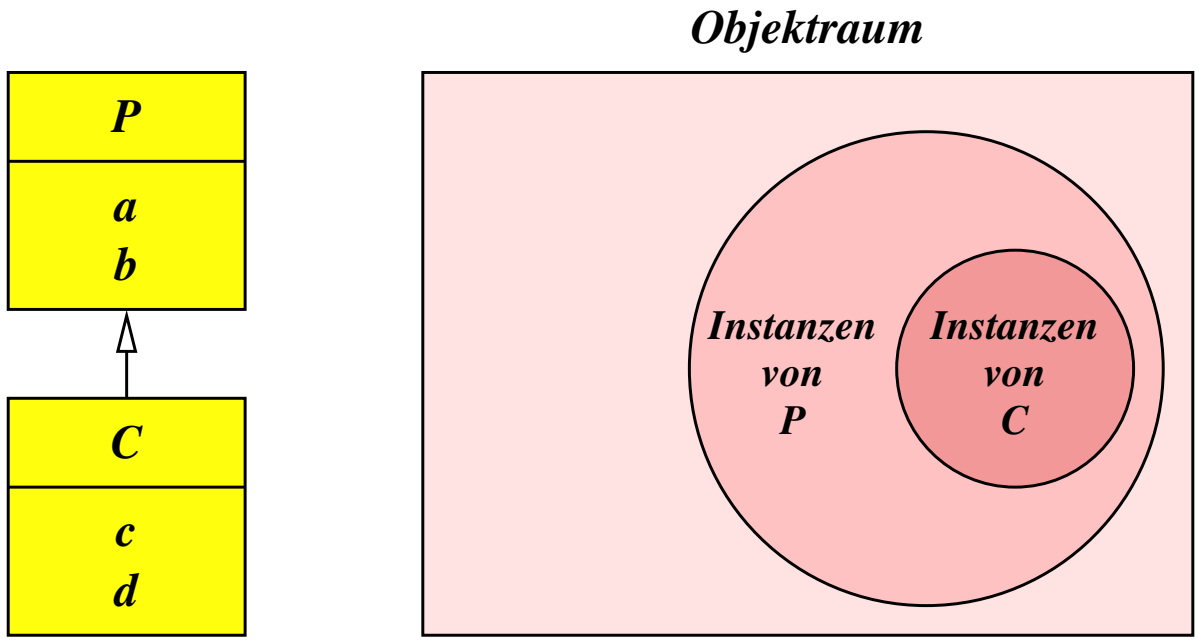
- Einige Klienten benötigen nur allgemeine Listen, die alles akzeptieren, was ihnen gegeben wird. Diese Klienten interessieren sich nicht für die Reihenfolge, in der die Listenelemente später entnommen werden.
- Der Vertrag für *List* spezifiziert, dass *pop()* bei Einhalten der Vorbedingung einer nicht-leeren Liste irgendein zuvor eingefügtes Element zurückliefert, das bislang noch nicht zurückgegeben worden ist. Die Reihenfolge selbst bleibt undefiniert.
- *Queue* erweitert diesen Vortrag dahingehend, dass als Ordnung die ursprüngliche Reihenfolge des Einfügens gilt (FIFO).
- *Stack* hingegen erweitert diesen Vertrag mit der Spezifikation, dass *pop()* das zuletzt eingefügte Element zurückzuliefern ist, das bislang noch nicht zurückgegeben wurde (LIFO).
- Erweiterungen sind jedoch verpflichtet, in jedem Falle den Vertrag der Basisklasse einzuhalten. Entsprechend dürfen Verträge nicht durch Erweiterungen abgeschwächt werden.
- Die Einhaltung dieser Regel stellt sicher, dass ein Objekt des Typs *Stack* überall dort verwendet werden darf, wo ein Objekt des Typs *List* erwartet wird.

- Vererbung ist im Bereich der OO-Techniken eine Beziehung zwischen Klassen, bei denen eine *abgeleitete Klasse* den Vertrag mitsamt allen Signaturen von einer *Basisklasse* übernimmt.
- Da in der Mehrzahl der OO-Sprachen Klassen mit Typen kombiniert sind, hat die Vererbung zwei Auswirkungen:
  - Kompatibilität: Instanzen der abgeleiteten Klasse dürfen überall dort verwendet werden, wo eine Instanz der Basisklasse erwartet wird.
  - Gemeinsamer Programmtext: Die Implementierung der Basisklasse kann teilweise von der abgeleiteten Klasse verwendet werden. Dies wird für jede Methode einzeln entschieden. Einige OO-Sprachen (einschließlich C++) ermöglichen den gemeinsamen Zugriff auf ansonsten private Datenfelder zwischen der Basisklasse und der abgeleiteten Klasse.

- Die Komplexität dieser Beziehung kann 1:* (*einfache Vererbung*) oder *:* (*mehrfache Vererbung*) sein.
- C++ unterstützt mehrfache Vererbungen.
- Java unterstützt nur einfache Vererbungen, bietet aber zusätzlich das typen-orientierte Konzept von Schnittstellen an.
- In C++ kann die Schnittstellen-Technik von Java auf Basis sogenannter abstrakter Klassen erreicht werden. In diesem Falle übernehmen Basisklassen ohne zugehörige Implementierungen die Rolle von Typen.

Vertragsraum vs. Objektraum





- Erweiterbarkeit / Polymorphismus: Neue Funktionalität kann hinzugefügt werden, ohne bestehende Klassen zu verändern, solange die neuen Klassen sich als Erweiterung bestehender Klassen formulieren lassen.
- Wiederverwendbarkeit: Für eine Serie ähnlicher Anwendungen kann ein Kern an Klassen definiert werden (*framework*), die jeweils anwendungsspezifisch erweitert werden.
- Verbergung (*information hiding*): Je allgemeiner eine Klasse ist, umso mehr verbirgt sie vor ihren Klienten. Je mehr an Implementierungsdetails verborgen bleibt, umso seltener sind Klienten von Änderungen betroffen und der Programmtext des Klienten bleibt leichter verständlich, weil die vom Leser zu verinnerlichenden Verträge im Umfang geringer sind.

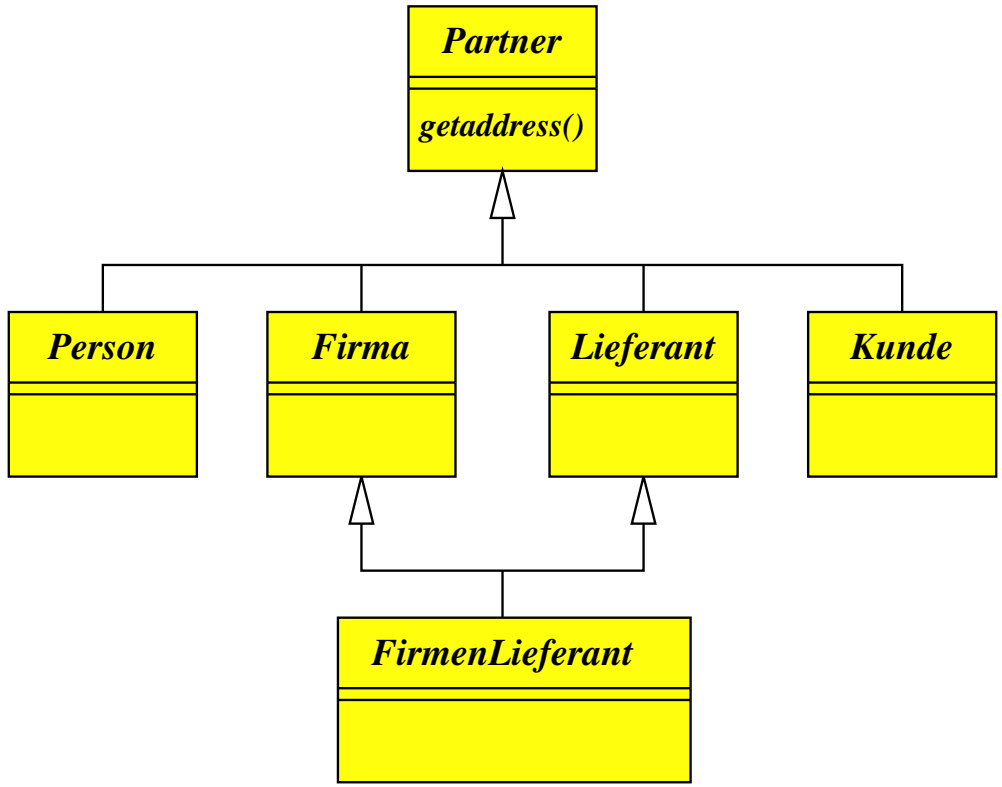
Vererbung sollte genutzt werden, wenn

- mehrere Implementierungen mit einer gemeinsamen Schnittstelle auskommen können, wenn
- Rahmen (*frameworks*) für individuelle Erweiterungen (*plugins*) sinnvoll sind und wenn
- sie zur Schaffung einer sinnvollen Typhierarchie dient, die die statische Typsicherheit erhöht.

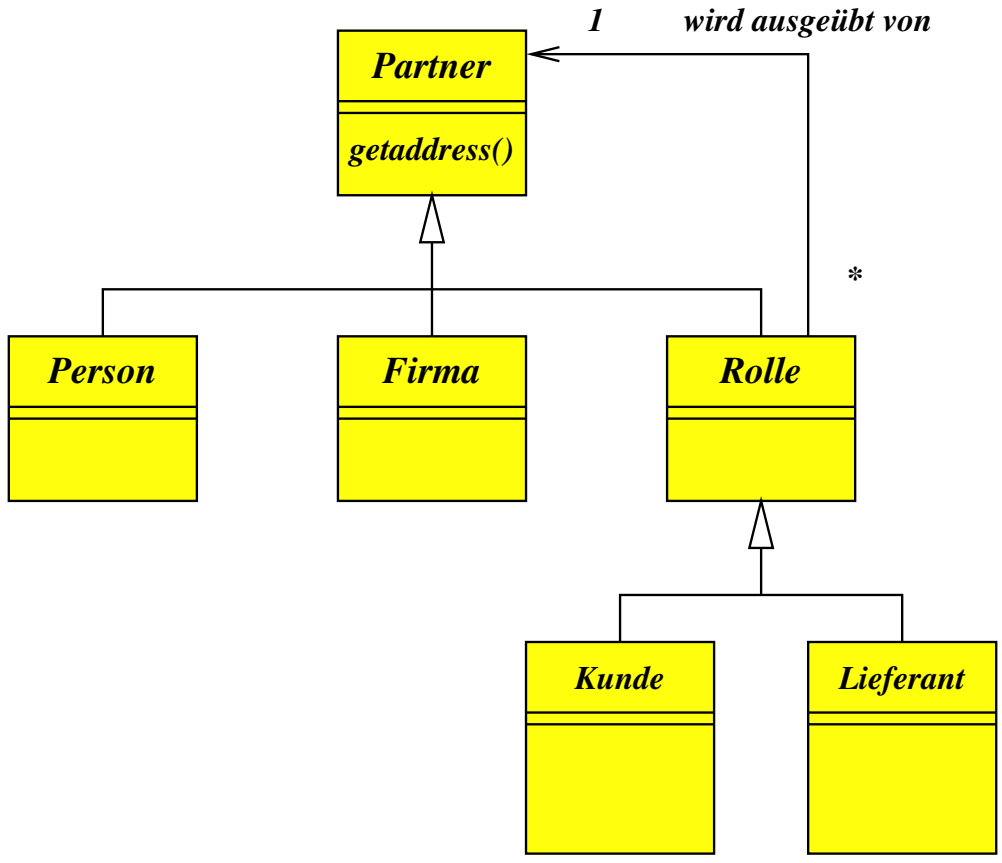
Vererbung sollte **nicht** genutzt werden, um

- bereits existierenden Programmtext wiederzuverwenden, wenn es sich dabei nicht um eine strikte *is-a*-Beziehung im Sinne einer sauberen Vertragshierarchie handelt oder um
- Objekte in ein hierarchisches Klassensystem zu zwängen, wenn diese bzw. deren zugehörigen realen Objekte die Einordnung im Laufe ihrer Lebenszeit verändern können.

Beispiel für schlechte Vererbung



- Die Rollenverteilung (z.B. als Lieferant oder Kunde) ist statisch und die Zahl der Kombinationsmöglichkeiten (und der entsprechend zu definierenden Klassen) explodiert.



- Ein Partner-Objekt kann während seiner Lebenszeit sowohl die Rolle eines Kunden oder auch eines Lieferanten übernehmen.
- Dieses Pattern entspricht dem Decorator-Pattern aus dem Werk von Gamma et al.