Klassendiagramme

Class diagrams

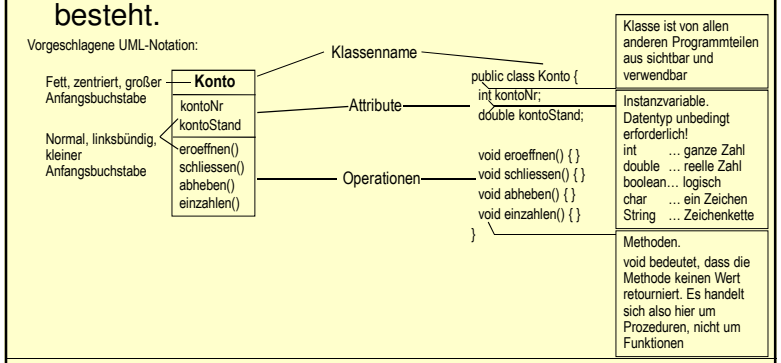
Aufgabe von Klassendiagrammen

Identifizieren die (realen und konzeptuellen) Objekte, die für das Modell von Bedeutung sind, deren grundlegende Eigenschaften sowie die Beziehungen zwischen verschiedenen Objekten.

Klassendiagramme sind statisch. Sie zeigen nur mögliche Beziehungen und Eigenschaften auf, machen aber keine Aussage, welche der Beziehungen zu verschiedenen Zeitpunkten existieren, bzw. relevant sind, und wie sich die Eigenschaften im Laufe der Zeit entwickeln. Sie gehören daher, wie die Use Case Diagramme zu den UML Diagrammen, die die statische Struktur (static structure) veranschaulichen.

Klassen

Werden durch ein Rechteck dargestellt, das im Normalfall aus drei Abteilungen (compartments) besteht.



Klassen: alternative Notationen (1)

Für die konkrete Darstellung nicht benötigte Compartments können unterdrückt werden.

Ein nicht vorhandenes Compartment bedeutet nicht, dass es auch leer sein muss. (Konto hat sehr wohl Attribute, diese sind aber im aktuellen Kontext nicht interessant.)

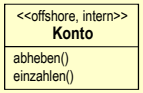
Ebenso müssen nur die relevanten Einträge angeführt werden. Es gibt also noch mehr (hier nicht interessante) Operationen, die aber unterdrückt wurden.

Diese Form der Darstellung unterdrückter Compartments wird nicht mehr empfohlen!

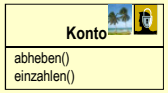
Beliebig viele zusätzliche Compartments können definiert werden. Der Inhalt ist im Wesentlichen freier Text mit einem optionalen Titel.

Wird z.B. verwendet um die Verantwortlichkeiten (responsibilities) einer Klasse zu dokumentieren, zur Beschreibung der Art und Behandlung von möglichen Fehlern (exceptions) und Ereignissen (events) oder verschiedener Varianten (variations) der Klasse.

Klassen: alternative Notationen (2)

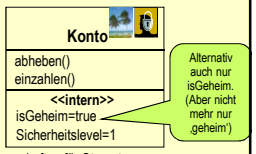


(Selbst definierte) Stereotypen können der Klasse zusätzliche Semantik verleihen. Mehrere Stereotypen werden durch Beistrich getrennt hintereinander geschrieben. Optional können auch mehrere guillemets verwendet werden. Diese Darstellung ist notwendig, wenn Stereotypen und Keywords gemischt werden, z.B.: <<actor>><<offshore>>



Stereotypen können durch eventuell definierte Icons ersetzt werden. Die Möglichkeit, die gesamte Box durch das Icon des Stereotyps zu ersetzen, besteht nur, wenn nur ein Stereotyp angewendet werden soll (Etwa bei der Verwendung des Strichmännchens bei den Aktoren).

Von UML vordefinierte Icons werden durch das Icon eines zusätzlichen Sterotyps ersetzt (vgl. Geschäftspartner auf dem Beiblatt).



Eigenschaften für Stereotypen können in optionalen Compartments vorgegeben werden. Dabei ist nötigenfalls für jeden Stereotyp ein eigenes Compartment zu verwenden.

Für Attribute vom Typ Boolean kann man die Konvention verwenden, dass der Wert true ist, wenn das Attribut angeführt wird, false sonst. Die früher übliche Notation, die für Namen der Form ‚isName' festlegte, dass statt {isName=true} auch nur {name} verwendet werden kann, wird in den Dokumenten zu UML 2 nicht mehr erwähnt.

Notationen für Attribute (1)

Man kann nicht nur den Namen eines Attributs definieren, sondern auch verschiedene Details festlegen. Dazu dient folgende Syntax:

<attribut> ::= [<visibility>] ['/'] <name> [':' <prop-type>] ['[' <multiplicity> ']'] ['=' <default>]
['{' <prop-modifier> {',' <prop-modifier>} '}']

<visibility> ::= '+' | '-' | '#' | '~'
// + : public (Zugriff für alle Klassen)
// - : private (Zugriff nur für diese Klasse),
// # : protected (Zugriff für dies Klassen und Klassen, die davon erben),
// ~ : package (Zugriff für alle Klassen im gleichen Package)

/ bedeutet abgeleitetes Attribut (derived attribute). Das bedeutet, der Wert dieses Attributs kann aus anderen Attributen jederzeit berechnet werden. Z.B.: isUeberzogen könnte aus dem kontoStand hergeleitet werden. (true, falls kontoStand < 0, false sonst). Erspart Rechenaufwand für oft benötigte Attributwerte. Normalerweise sind solche Attribute readOnly. Können sie aber verändert werden, so ist dafür Sorge zu tragen, dass die Werte, von denen das Attribut abhängt auch entsprechend verändert werden.

<name> ist der Name des Attributs

[ ] bezeichnen optionale Elemente. { } heißt beliebig oft wiederholen. | legt eine Auswahl fest.
In " eingeschlossene Zeichen sind direkt zu übernehmen. // ist der Beginn eines Kommentars.

Notationen für Attribute (2)

<attribut> ::= [<visibility>] ['/'] <name> [':' <prop-type>] ['[' <multiplicity> ']'] ['=' <default>]
['{' <prop-modifier> {',' <prop-modifier>} '}']

<prop-type>
*ist der Typ des Attributs. Hier können die primitiven Typen integer, boolean, string und unlimited natural (natürliche Zahlen mit 0 und * für unendlich) verwendet werden, aber auch vor- und selbstdefinierte Typen (Klassen). Meist werden einfach die Datentypen der zur Implementierung verwendeten Programmiersprache verwendet.*

<multiplicity>
legt die Anzahl der Elemente fest. Default ist 1.
Syntax ist <untergrenze>..<obergrenze>.
Falls beide Werte gleich sind, kann einfach nur dieser Wert angegeben werden (z.B.: 5).
*Als Obergrenze ist auch ' * ' erlaubt.*
Beispiele:
[1..10] von 1 bis 10, [5] genau 5,
[3..] mindestens 3, [0..3] höchstens 3,*
[0..] oder auch nur [*] beliebig viele*

<default> ist ein Ausdruck, der den Defaultwert (die Defaultwerte) für das Attribut angibt.

Notationen für Attribute (3)

<prop-modifier> ::= 'readOnly' | 'union' | 'subsets' <property-name> | 'redefines' <property-name> | 'ordered' | 'unique' | <prop-constraint>

readOnly
bedeutet, dass dieses Attribut nur einmal initialisiert wird und dann nicht mehr verändert werden kann. Wann die Initialisierung stattfindet, legt UML nicht fest. Wird allerdings ein Defaultwert angegeben, so gilt das bereits als Initialisierung. (Entspricht dem Konzept von Konstanten in Programmiersprachen).

union
bedeutet, dass dieses (zumeist abgeleitete) Attribut die Vereinigung von mit subsets spezifizierten Untermengen ist.

subsets <property-name>
bedeutet, dass dieses Attribut eine Untermenge der im Attribut property-name enthaltenen Werte enthält

redefines <property-name>
legt fest, dass dieses Attribut nur ein anderer Name (alias) für das Attribut <property-name> ist. Wird zumeist in Verbindung mit Vererbung verwendet.

Notationen für Attribute (4)

<prop-modifier> ::= 'readOnly' | 'union' | 'subsets' <property-name> | 'redefines' <property-name> | 'ordered' | 'unique' | <prop-constraint>

ordered

gibt für Attribute mit einer Multiplizität größer 1 an, dass die einzelnen Werte geordnet (sortiert) zugreifbar sind. Welches Sortierkriterium verwendet wird, gibt UML nicht vor. Es ergibt sich zumeist aus dem Kontext (z.B. alphabetisch für Strings, numerisch für Zahlen, etc.)

unique

gibt für Attribute mit einer Multiplizität größer 1 an, dass die einzelnen Werte eindeutig sein müssen. Jeder Wert darf also (innerhalb diese Attributs in einem Objekt) höchstens einmal auftreten. In verschiedenen Objekten darf der gleiche Wert natürlich sehr wohl wiederholt auftreten

<prop-constraint>

gibt zusätzliche Einschränkungen an, die erfüllt sein müssen. Entspricht einem logischen Ausdruck in OCL, einer Programmiersprache (z.B.: Java) oder natürlicher Sprache. Dieser Ausdruck muss zu jedem Zeitpunkt den Wert true ergeben. Constraints können mit einem Namen versehen werden, indem man dem Ausdruck den Namen mit einem Doppelpunkt voranstellt. Z.B.: kontoGedeckt: not(isUeberzogen)

Notationen für Operationen

Für Operationen (Methoden in Java) gilt eine ähnliche Syntax:

<operation> ::= [<visibility>] <name> '(' [<parameter-list>] ')' [':' [<return-type>] ['{' <oper-property> [',' <oper-property>]* '}']]

<name>

ist der Name der Operation

<return-type>

ist der Typ des retournierten Wertes. Für Prozeduren sollte der return-type entfallen, oft wird auch (in Anlehnung an die verwendete Programmiersprache) void verwendet.

<oper-property>::='redefines' <oper-name> | 'query' | 'ordered' | 'unique' | <oper-constraint>

redefines <oper-name> bedeutet, dass die ererbte Operation oper-name neu definiert wird.

query bedeutet, dass die Operation den Zustand des Systems nicht verändern darf.

ordered bedeutet, dass die Werte im gelieferten Returnwert sortiert sind.

unique bedeutet, dass jeder Wert im gelieferten Returnwert nur einmal auftreten darf.

<oper-constraint> gibt zusätzliche Bedingungen an, die erfüllt sein müssen. Die Syntax dieser Constraints ist analog zu den Constraints bei Attributen.

*<parameter-list>::=<parameter> [',<parameter>]**

Notationen für Operationen: Parameter

<parameter-list>::= <parameter> {','<parameter>}

<parameter> ::= [<direction>] <parameter-name> ':' <type-expression>['['<multiplicity>']'] ['=' <default>] ['{' <parm-property> [',' <parm-property>] '}']*

<direction>::= 'in' | 'out' | 'inout' | 'return'

return bietet eine optionale Möglichkeit, den Returnwert zu definieren. Default ist ‚in' (Das ist in Java auch die einzig mögliche Parameterart).

<parameter-name>

ist der Name des Parameters

<type-expression>

gibt den Typ des Parameters an

<mutliplicity>

Multiplizität analog zu Attributen

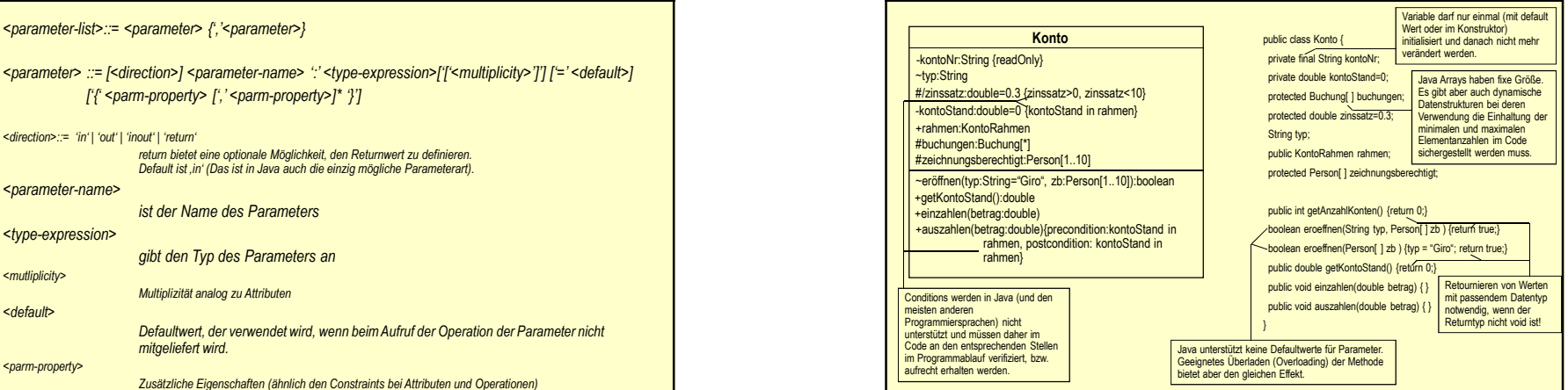
<default>

Defaultwert, der verwendet wird, wenn beim Aufruf der Operation der Parameter nicht mitgeliefert wird.

<parm-property>

Zusätzliche Eigenschaften (ähnlich den Constraints bei Attributen und Operationen)

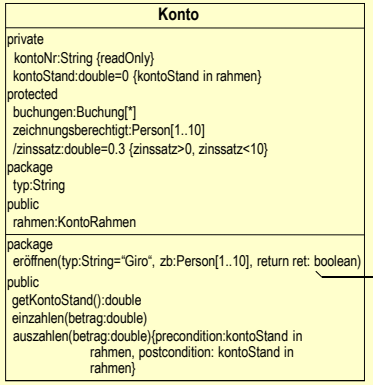
Beispiele



13

Alternative Darstellung

Alternativ können Attribute und Methoden nach Sichtbarkeit gruppiert werden:



Eher unübliche Darstellung für einen Returnwert.

14

Statische Attribute und Methoden

Jede Methode wird genau immer für ein Objekt (eine Instanz) der Klasse aufgerufen. Jede Instanz hält eigene Kopien der jeweiligen Attribute (Instanzvariablen) vor.

Manchmal ist es vorteilhaft, eine Methode aufrufen zu können, ohne sich auf ein bestimmtes Objekt zu beziehen, bzw. Attribute zu haben, die sich auf alle Objekte der Klasse gleichzeitig beziehen.

Dafür gibt es statische Attribute und Methoden (auch Klassenvariablen und Klassenmethoden) genannt.

Innerhalb statischer Methoden darf nur auf statische Attribute und Methoden der Klasse zugegriffen werden!


```
public class Konto {
  private static int anzahlKonten=0;
  public static int getAnzahlKonten() {return anzahlKonten;}
}
```

anzahlKonten gibt die Anzahl der insgesamt aktuell vorhandenen Konten an. Diese Anzahl muss beim Anlegen eines Kontos immer erhöht und beim Auflösen wieder vermindert werden.

15

Konstruktor und Destruktor

In vielen Fällen müssen beim Erstellen eines neuen Objekts (Instanziierung) Aktionen gesetzt werden, um das Objekt in einen gültigen Anfangszustand zu bringen (Initialisierung) und den Umgebungszustand entsprechend zu aktualisieren (z.B.: Erhöhen des Zählers für Konten, wie auf der vorhergehenden Folie).

Analog dazu muss am Ende des Lebenszyklus eines Objekts (Destruktion) wieder ‚aufgeräumt' werden.

Für diese Zwecke stehen in Programmiersprachen Konstruktoren und Destruktoren zur Verfügung.

Konstruktor wird für jedes neu instantiierte Objekt automatisch aufgerufen. Kann für verschiedene Parameterkombinationen überladen werden. Die parameterlose Version wird Defaultkonstruktor genannt.

Heißt wie die Klasse und hat keinen Returntyp. (Kann in Spezialfällen auch nicht public sein.)

Im konkreten Fall wird die (anschließend nicht mehr änderbare) kontoNr generiert und anzahlKonten erhöht.

```
public class Konto {
  private static int anzahlKonten=0;
  private final String kontoNr

  public Konto() {kontoNr=getNextKontoNr(); anzahlKonten=anzahlKonten+1;}
  public static int getAnzahlKonten() {return anzahlKonten;}
  protected void finalize() {anzahlKonten=anzahlKonten-1;}
}
```

Destruktor wird aufgerufen, wenn das Objekt zerstört wird. Muss immer genau so aussehen.

Da in Java der genaue Zeitpunkt der Zerstörung von Objekten nicht definiert ist (garbage collection), werden Destruktoren hier nur selten verwendet.

Im konkreten Fall könnte ein schon geschlossenes Konto noch immer irgendwo im Speicher ‚herumliegen' und würde in anzahlKonten dementsprechend mitgezählt.

16

Beziehungen (relationships)

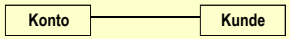
Zwischen Klassen (und Objekten) können vielfältige Beziehungen bestehen. Diese werden in UML Diagrammen durch entsprechend ausgeführte und dekorierte Linien visualisiert:

- Assoziationen. Verschiedene Dekorationen möglich
- Generalisierung (Vererbung)
- Realisierung (realization; wird bei den Schnittstellen behandelt)
- Abhängigkeit (dependency; z.B. bei <<extend>> und <<include>> bei Use Cases)

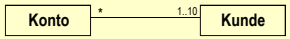
17

Assoziationen

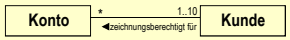
Assoziationen zwischen Klassen zeigen an, dass Objekte der einen Klasse zur Laufzeit mit Objekten der anderen Klasse (irgendwann) in einer Beziehung stehen.



Ein Konto steht mit einem Kunden in Beziehung



Multiplizitäten: Ein Kunde kann beliebig viele Konten haben, ein Konto kann mit mindestens einem und maximal 10 Kunden in Beziehung stehen. (Default 1. Keine eckigen Klammern wie bei Attributen oder Operationen nötig.)

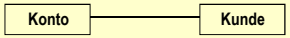


Name und Leserichtung (beides optional).

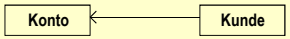
18

Assoziationen: Navigierbarkeit

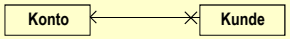
Mit Hilfe von Pfeilen kann die Navigierbarkeit angezeigt werden. Sind gar keine Pfeile angegeben (wie auf der vorhergehenden Folie), dann kann in beiden Richtungen navigiert werden. Sonst nur in die Richtung der Pfeile.



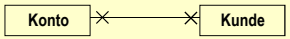
Kunde kann seinem Konto zugeordnet werden und umgekehrt (selten auch, falls gar keine Zuordenbarkeit besteht; UML legt nicht fest, ob diese Schreibweise als ⟷ oder als ⤫—⤫ interpretiert werden soll)



Konto des Kunden kann gefunden werden, aber der Kunde zum Konto nicht (wenn es sich etwa um ein geschlossenes Konto handelt und die Daten bei der Bank nicht mehr gehalten werden, der Kunde aber noch Kontoauszüge besitzt).



Navigierbarkeit kann explizit verhindert werden (redundant)

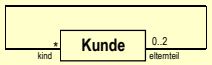


Möglich, aber in der Praxis kaum von Bedeutung

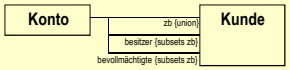
19

Assoziationen: Rollen

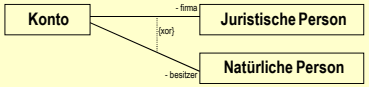
Rollennamen können das Verständnis erleichtern, besonders wenn mehrere Beziehungen zur gleichen Klasse bestehen



Ein Kunde kann beliebig viele Kinder haben (die auch Kunden sind). Umgekehrt können höchstens zwei Elternteile auch Kunden sein. UML verlangt nicht, dass die beiden assoziierten Objekte verschieden sind. Diese Notation erlaubt also auch Kunden, die Kinder von sich selbst sind. Durch zusätzliche Constraints ließe sich das verhindern.



Zeichnungsberechtigter kann entweder Besitzer oder Bevollmächtigte sein. Es sind die gleichen Constraints, wie bei Attributen erlaubt.
(Alternative graphische Darstellung zu drei separaten Linien.)

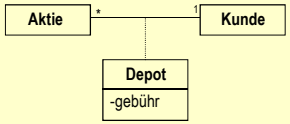


xor ist ein vordefinierter Constraint, der festlegt, dass nur jeweils eine der beiden Assoziationen zu einem bestimmten Zeitpunkt ausgeprägt sein kann.
Für die Rollen können, wie für Attribute, Sichtbarkeiten festgelegt werden.

20

Attributierte Assoziation

Oft müssen zu einer Assoziation zusätzliche Daten (Attribute) verwaltet werden. Man behilft sich mit einer so genannten Assoziationsklasse:

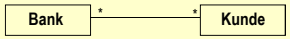


Die Aktien gehören dem Kunden, weil sie in seinem Depot verwaltet werden.

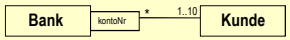
21

Qualifizierte Assoziationen

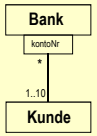
Auswahl aus möglichen Objekten mittels Qualifizierung (qualifier):



Bank mit beliebig vielen Kunden, die ihrerseits wieder beliebig vielen Banken nutzen.



Auswahl der zeichnungsberechtigten Kunden mittels kontoNr. Multiplizität ändert sich!

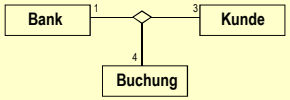


Alternative Darstellung

22

N-äre Assoziationen

Eine Assoziation kann auch mehr als nur zwei Elemente verbinden:



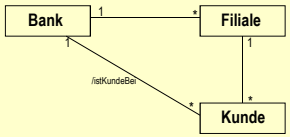
Beispiel einer ternären Assoziation. Interpretation der Multiplizitäten:
Ein Kunde hat bei einer Bank vier Buchungen.
Eine Buchung bei einer Bank hat betrifft drei Kunden.
Eine Buchung eines Kunden ist einer Bank zugeordnet.

Dies kann für beliebig viele Elemente verallgemeinert werden. Alle zuvor beschriebenen Eigenschaften (Navigierbarkeit, Rollen, Attributierungen und Qualifizierungen) können auch bei n-ären Assoziationen verwendet werden.

23

Abgeleitete Assoziationen

Derived associations (Ähnlich wie derived attributes):

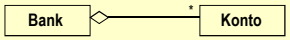


Die Assoziation „istKundeBei“ kann aus den beiden anderen hergeleitet werden. (Sollen Kunden mehrere Banken nutzen dürfen, dann muss auch die Multiplizität bei „Filiale“ passend geändert werden.)

24

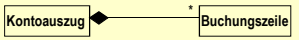
Aggregationen und Kompositionen

Eine Aggregation ist eine etwas stärkere Form der Assoziation, bei der typischerweise ein Art ‚besitzt‘ Relation beschrieben wird.



Darstellung mittels leerer Raute auf der Seite des Besitzers

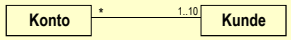
Die Komposition ist noch stärker und beschreibt eine Teil-Ganzes-Beziehung oder ‚besteht aus‘ Relation



Darstellung mittels voller Raute auf der Seite des Ganzen

Assoziationen und Attribute

Assoziationen sind semantisch äquivalent zu entsprechenden Attributen und werden in Programmiersprachen meist auch als Attribute implementiert. Eigenschaften der Assoziationen, die von der Programmiersprache nicht direkt unterstützt werden (z.B. Constraints) müssen im Code der entsprechenden Methoden berücksichtigt werden.




```
public class Konto {
  private Kunde[ ] zeichnungsberechtigte;
  …
}

public class Kunde {
  private Konto[ ] konten;
}
```

Der Programmcode hat die Konsistenz sicher zu stellen, d. h., dass zu ‚jedem Zeitpunkt' die jeweils entsprechenden Werte in beiden Feldern gespeichert sind.
Um Navigierbarkeit in einer Richtung zu unterbinden, wird das entsprechende Attribut einfach entfernt.

Objektdiagramme

Object diagrams

Darstellung von Instanzen

Für viele Anwendungen ist die Darstellung der Beziehungen zwischen den Klassen alleine zu grob. Man kann auch Beziehungen zwischen einzelnen Objekten (Instanzen) darstellen:

WichtigsterKunde:Kunde	Ein bestimmtes Objekt namens WichtigsterKunde der Klasse Kunde
WichtigsterKunde	Klasse kann entfallen, wenn sie aus dem Kontext klar ist
:Kunde	Irgendein Objekt der Klasse Kunde
WichtigsterKunde Kundennummer=0815 Name= " Fred Feuerstein"	Für das Objekt können Attributwerte vorgegeben werden