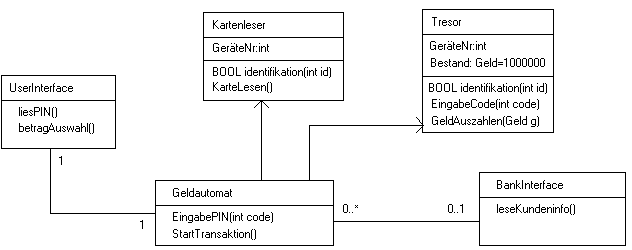
**Modul 320**

***- Klassenbeziehungen I -***



**Inhaltsverzeichnis:**

[1 Die lose Beziehung (use- & instanziiert) 2](#_Toc153530191)

[2 Die Assoziationsbeziehung 3](#_Toc153530192)

[2.1 Unidirektionale-Assoziationen 4](#_Toc153530193)

[2.2 Bidirektionale Assoziationen 5](#_Toc153530194)

[2.3 Gerichtete 1:n Beziehungen 6](#_Toc153530195)

[2.4 Bidirektionale 1:n Beziehungen 7](#_Toc153530196)

[2.5 Ein kombiniertes Beispiel 8](#_Toc153530197)

[2.6 Bidirektionale Beziehungen automatisch setzen 9](#_Toc153530198)

[2.7 Aggregation und Komposition: Spezialfälle einer Assoziation 10](#_Toc153530199)

**Inhalt / Lernziele:**

Dieser Block repetiert und vertieft zuerst die im Modul 404 erlernten Kenntnisse bezüglich der Erstellung eigener Klassen.

Kernstück ist das Kennenlernen der losen- und der Assoziations-Beziehungen inkl. deren Untergruppen Aggregations- und Kompositionsbeziehung, zwischen Klassen. Ziel ist, dass sie ein gegebenes UML-Klassendiagramm, welches die oben erwähnten Beziehungstypen enthält, in Programmcode umsetzen können. Genauso müssen sie auch bestehenden Programmcode mittels UML-Klassendiagramm visualisieren können.

Die Kapitel 1-3 enthalten die Theorie, das Kapitel 4 enthält Übungen.

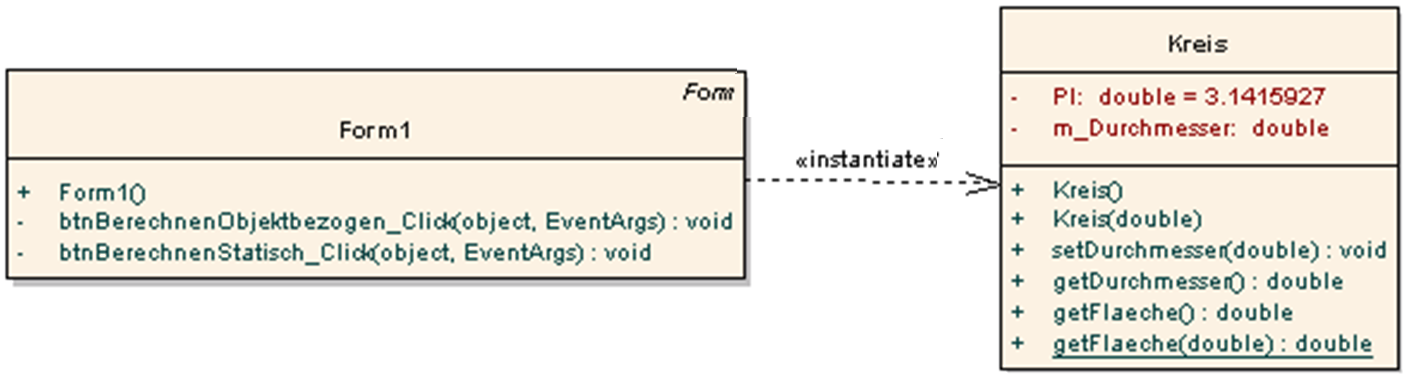
# 

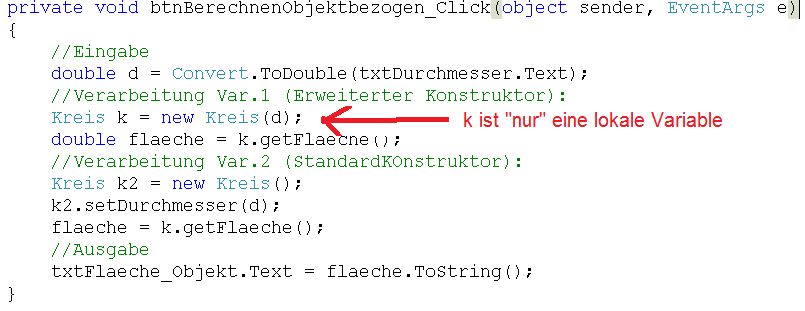
# Die lose Beziehung (use- & instanziiert)

Es existieren lose Beziehungen zwischen Klassen. Von einer losen Beziehung spricht man, wenn wir keine Referenz (eine Speichersdresse eines Objektes) als Membervariable speichern. Man unterscheidet zwei wesentliche Typen: «use» und «instanziiert».

**Typ 1 «instanziiert» (Referenz nur temporär als lokale Variable verfügbar):**

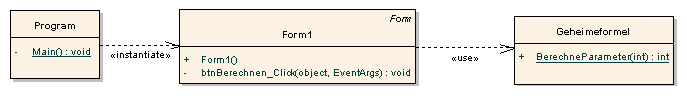
Wenn wir ein Objekt erstellen, welches ein anderes Objekt verwendet, ohne dessen Referenz als Membervariable zu speichern, handelt es sich um eine lose «inszanziiert» Beziehung.

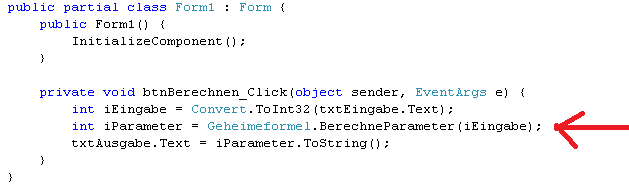




**Typ 2 «use» (Zugriff via Klassenname auf ein statisches Element einer anderen Klasse):**

Wenn wir auf ein statisches Element einer anderen Klasse zugreifen, handelt es sich um eine lose «use» Beziehung.





# Die Assoziationsbeziehung

Prinzipiell sprechen wir von einer Assoziationsbeziehung, wenn ein Objekt einer Klasse, eine Referenz (die Speicherstelle eines anderen Objektes) als Member­variable speichert.

Die Assoziationsbeziehungen kann man nach verschiedenen Gesichtspunkten kategorisieren:

Nach ihrer Richtung:

* unidirektionale (gerichtete) Assoziation
* bidirektionale Assoziation

Nach der Mengenangabe:

* 1:1- Assoziation (ein Objekt kann ein anderes Objekt referenzieren)
* 1:n- Assoziation (ein Objekt kann ein oder mehrere andere Objekte referenzieren)
* N:m-Assoziation (Objekte können sich gegenseitig mehrfach referenzieren)

Nach der Wirkung:

* Assoziation
* Aggregation (Spezialfall der Assoziation)
* Komposition (Spezialfall der Aggregation)

Diese Kategorien kommen kombiniert vor. Man spricht also z.B. von einer bidirektionalen 1:n Aggregationsbeziehung.

Eine Assoziation modelliert Verbindungen zwischen Objekten einer oder mehrerer Klassen. Eine Assoziation modelliert stets Beziehungen zwischen Objekten, nicht zwischen Klassen. Es ist jedoch üblich von einer Assoziation zwischen Klassen zu sprechen, obwohl streng genommen die Objekte dieser Klassen gemeint sind.

[Abbildung](file:///\\LABSRV001\Klassen$\5_daten\Module\Bücher\galileocomputing_java2\kap_04.htm#bild1#bild1)***Abhängigkeiten, Multiplizität, Symbol \****

* Zwischen den beteiligten Klassen bestehen keine essenziellen Abhängigkeiten, d.h., ihre Objekte können unabhängig voneinander leben, erschaffen und zerstört werden.
* Die Multiplizität min..max an der Beziehung beschreibt die minimale bzw. maximale Anzahl von Objekten, mit der ein Objekt der gegenüberliegenden Klasse verbunden ist. Ist min gleich max, genügt eine Angabe.
* Der Stern \* steht für ein beliebiges oder nicht näher spezifiziertes Maximum. Steht nur ein Stern, ist dies die Abkürzung für 0..\*.

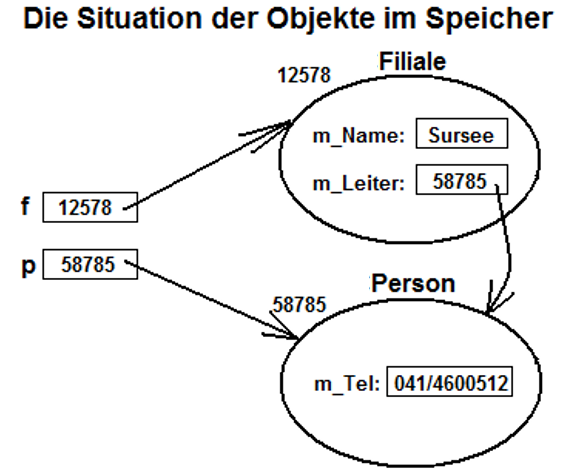
Die angegebene Multiplizität 4..18 steht für die Forderung, dass jedes Praktikum aus zumindest vier bis maximal 18 Studenten bestehen sollte. Ein Student kann an einer nicht näher spezifizierten Anzahl von Praktika teilnehmen.

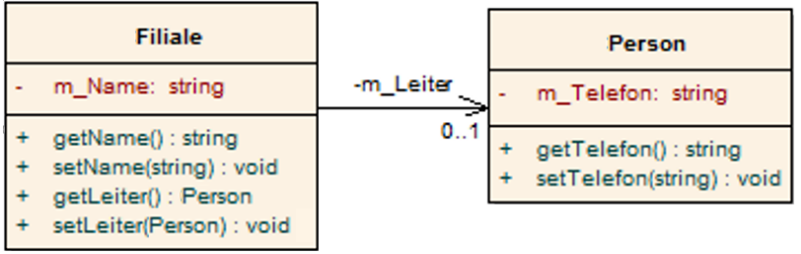
## Unidirektionale-Assoziationen

Wir sprechen von unidirektionalen Assoziationen, wenn nur ein Objekt der beiden in Beziehung stehenden Objekte die Adresse des anderen Objektes als Membervariable speichert.

Beispiel: Ein Kundenobjekt speichert die Adresse eines Bestellungsobjektes als Membervariable. Das Bestellungsobjekt hat aber im Gegenzug keine Kenntnis über diese Verbindung. Man kann es also nicht nach »seinem« Kundenobjekt fragen.

Im untenstehend dargestellten Fall weiss das Objekt der Klasse Filiale, wo sich das Objekt der Klasse Person befindet, da es dessen Adresse in einer Membervariablen gespeichert hat. Per UML erkennt man eine solche unidirektionale Beziehung am Pfeil ( 🡪 ). Die Mengenangabe 0..1 auf der Beziehung zeigt, dass die als Membervariable gespeicherte Referenz sowohl NULL (leer) als auch auf ein Objekt gesetzt sein kann.





class Program {

static void Main(string[] args) {

Filiale f = new Filiale();

f.setName("Sursee");

Person p = new CPerson();

p.setTelefon("041/4600512");

f.setLeiter(p);

}

}

public class Person {

private string m\_Tel;

public string getTelefon(){

return m\_Tel;

}

public void setTelefon(string value){

m\_Tel = value;

}

}

public class Filiale {

private string m\_Name;

private Person m\_Leiter =null;

public string getName() {

return m\_Name;

}

public void setName(string value) {

m\_Name = value;

}

public Person getLeiter(){

return m\_Leiter;

}

public void setLeiter(Person value){

m\_Leiter = value;

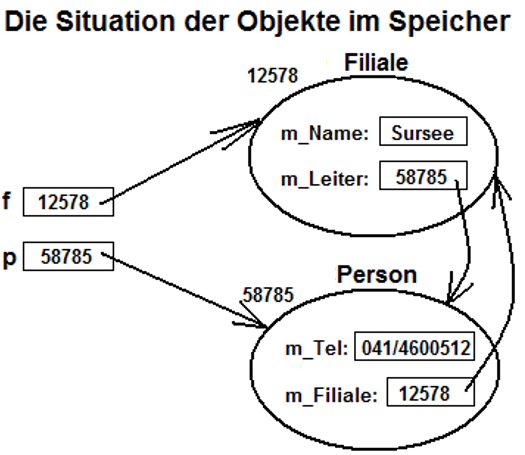
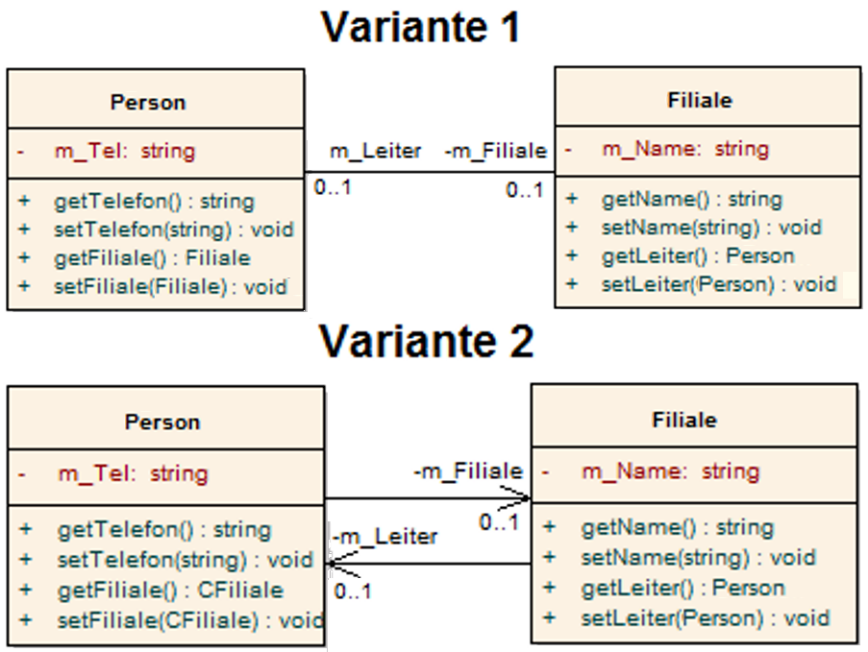
}

}

## Bidirektionale Assoziationen

Wir sprechen von bidirektionalen Assoziationen, wenn beide beteiligten Objekte die Adresse des jeweils anderen Objektes als Membervariable speichern.

Im untenstehend dargestellten Fall referenziert ein Filialenobjekt in einer Member-Variablen ein Personenobjekt. Das besagte Personenobjekt wiederum referenziert in einer Member-Variablen das Filialenenobjekt.



class Program {

static void Main(string[] args) {

Filiale f = new Filiale();

f.setName("Sursee");

Person p = new Person();

p.setTelefon("041/4600512");

f.setLeiter(p);

p.setFiliale(f);

}

}

public class Person {

private string m\_Tel;

private Filiale m\_Filiale =null;

public string getTelefon(){

return m\_Tel;

}

public void setTelefon(string value){

m\_Tel = value;

}

public Filiale getFiliale() {

return m\_Filiale;

}

public void setFiliale(Filiale value){

m\_Filiale = value;

}

}

public class Filiale {

private string m\_Name;

private Person m\_Leiter =null;

public string getName() {

return m\_Name;

}

public void setName(string value) {

m\_Name = value;

}

public Person getLeiter(){

return m\_Leiter;

}

public void setLeiter(Person value){

m\_Leiter = value;

}

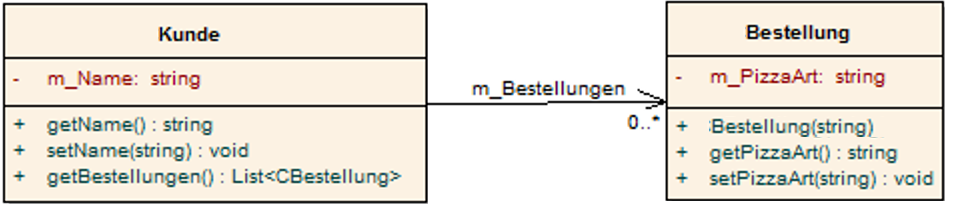
}

Die letzten beiden Zuweisungen in diesem Beispiel stellen die gegenseitige Verbindung zwischen den Objekten her. Beide sind damit indirekt zirkulär verknüpft. Dies Verknüpfung kann man programmiertechnisch so automatisieren, dass nur eine Zuweisung nötig ist. Das werden wir später in den Beispielen sehen.

Eine Objektreferenz ist letztlich nichts anderes als eine Speicheradresse und ein Objekt ein Speicherbereich auf dem Heap. Ohne Referenz existiert keine Möglichkeit mit einem Objekt zu interagieren.

## Gerichtete 1:n Beziehungen

Wenn ein Objekt mehrere Verweise auf andere Objekte enthalten kann, spricht man von einer 1:n-Beziehung zwischen den Klassen.



Da in den meisten Fällen die Zahl der Objekte auf der Child-Seite der Beziehung nicht festgelegt ist und auch während der Lebenszeit des Parent-Objekts wechselt, werden die Referenzen auf Child-Objekte am Besten in einem Datenfeld (Array) gespeichert.

public class Bestellung {

private string m\_PizzaArt;

public Bestellung(string PizzaArt){

setPizzaArt(PizzaArt);

}

public string getPizzaArt() {

return m\_PizzaArt;

}

public void setPizzaArt(string value){

m\_PizzaArt = value;

}

}

public class Kunde {

private string m\_Name;

private List<Bestellung> m\_Bestellungen = new List<Bestellung>();

public string getName(){

return m\_Name;

}

public void setName(string value) {

m\_Name = value;

}

public List<Bestellung> getBestellungen(){

return m\_Bestellungen;

}

}

public class Programm {

void Main(string[] args){

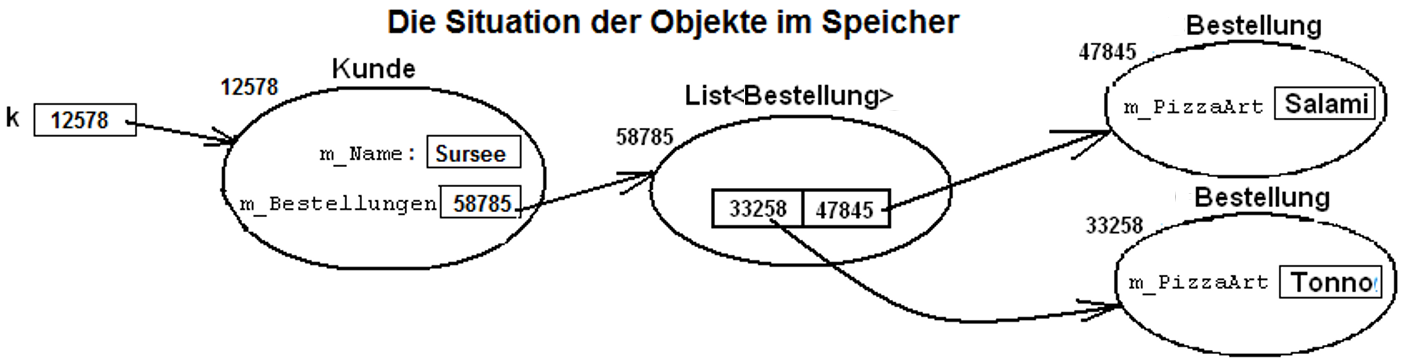
Kunde k = new Kunde();

k.getBestellungen().Add(new Bestellung("Tonno"));

k.getBestellungen().Add(new Bestellung("Salami"));

}

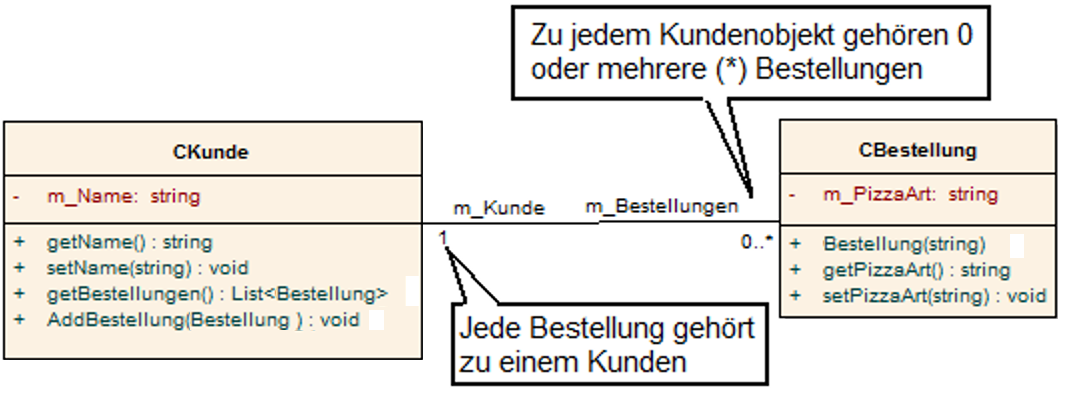
}



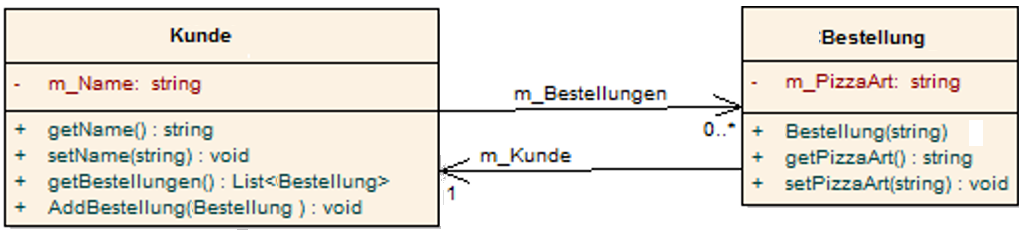
Das Feld m\_Bestellungen fasst alle Objekte der Child-Klasse (Bestellung) zusammen und unterstreicht damit den Listencharakter der Beziehung. Wenn sie die Zahl der angebundenen Child-Objekte sehr flexibel verändern wollen, ist ein statisches Datenfeld nicht die richtige Wahl, da diese eine feste Länge haben und entsprechend Speicherplatz in dieser Länge belegen, unabhängig davon, ob den Feldelementen Objekte zugewiesen sind oder nicht. Hinzu kommt, dass sie bei statischen Datenfeldern nur schwer neue Einträge jenseits der allokierten Länge hinzufügen oder Einträge herauslöschen können.

Bessere Lösungen stellen die oben verwendeten dynamischen Datenfelder dar. Diese werden meist in Form von Klassen zur Verfügung gestellt (z.B. die Klassen ArrayList oder List)

## Bidirektionale 1:n Beziehungen

Child-Objekte in einer bidirektionalen 1:n-Beziehung stehen in einer n:1 -Beziehung zu ihrem Parent. Da Objektreferenzen jedoch unidirektional sind, kann man diese Beziehung durch eine gerichtete 1:1-Be­ziehung modellieren.

Die beiden Varianten der Darstellungen sind gleichwertig.



Parent-Objekte enthalten eine Liste von Childobjekten und jedes Childobjekt hat eine Referenz auf das Parentobjekt:

public class Bestellung {

private string m\_PizzaArt;

private Kunde m\_Kunde;

public Bestellung(string PizzaArt){

setPizzaArt(PizzaArt);

}

public string getPizzaArt() {

return m\_PizzaArt;

}

public void setPizzaArt(string value){

m\_PizzaArt = value;

}

public Kunde getKunde() {

return m\_Kunde;

}

public void setKunde(Kunde value){

m\_Kunde = value;

}

}

public class Kunde {

private string m\_Name;

private List<Bestellung> m\_Bestellungen = new List<Bestellung>();

public string getName(){

return m\_Name;

}

public void setName(string value) {

m\_Name = value;

}

public void AddBestellung (Bestellung value){

m\_Bestellungen.Add(value);

**value.setKunde(this);**

}

}

public class Programm {

void Main(string[] args){

Kunde k = new Kunde();

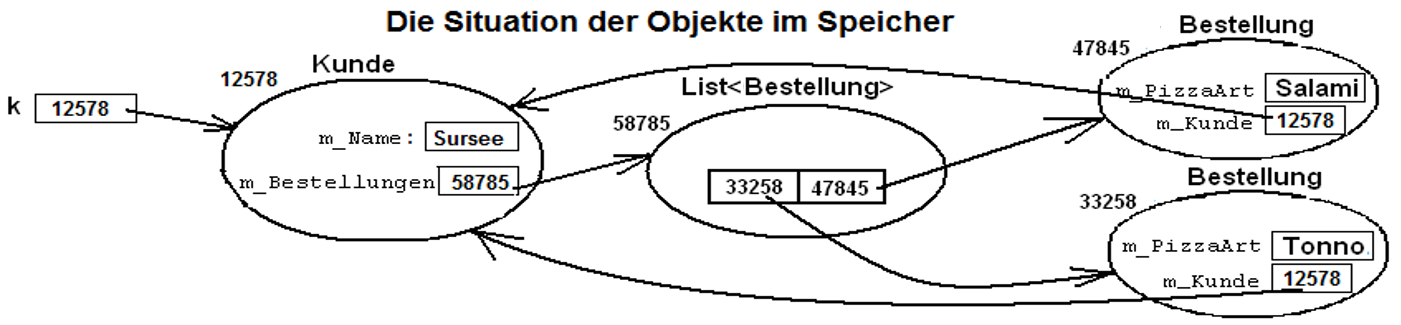
Bestellung b = new Bestellung("Tonno");

k.AddBestellung(b);

k.AddBestellung (new Bestellung("Salami"));

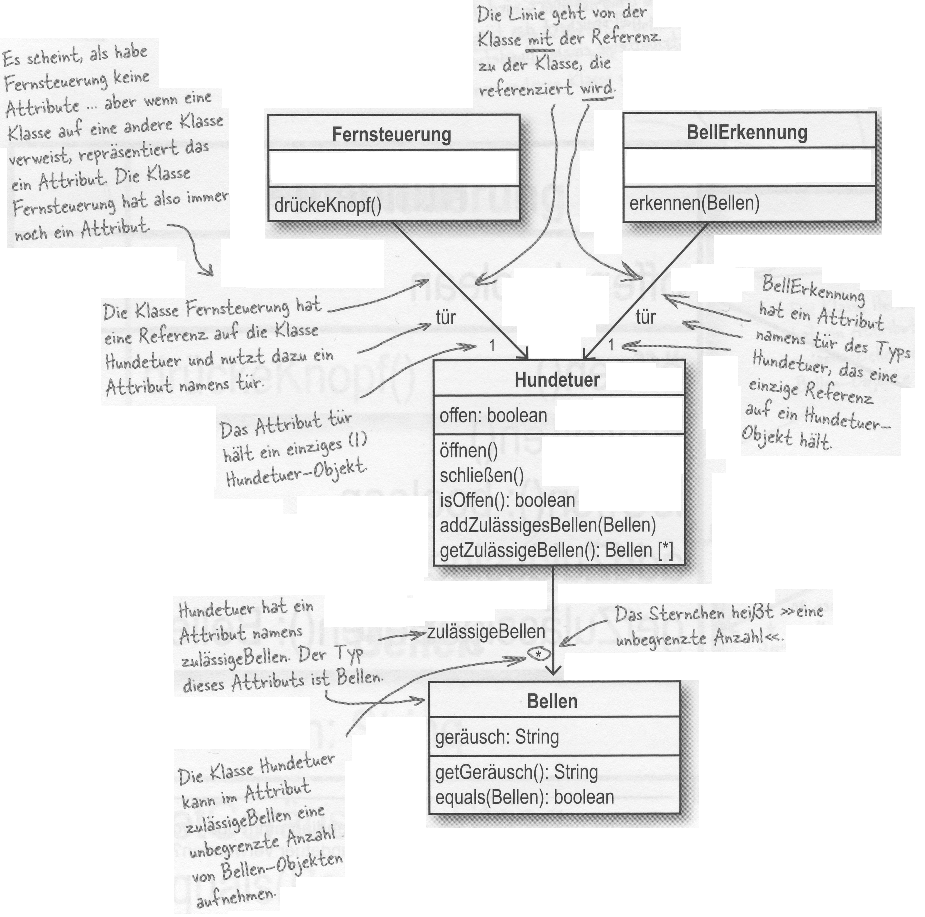
}

}

Selbstverständlich muss man auch hier darauf achten, das Child-Objekte immer auf den richten Parent verweisen. Die gegenseitige Verbindung der Partner sollte daher automatisch (via get- & set-Methoden) geschehen, wenn Sie ein Objekt auf einer beliebigen Seite in die Beziehung einbringen. Der oben dargestellte Programmcode berücksichtigt dies der Übersichtlichkeit wegen nicht.

## Ein kombiniertes Beispiel

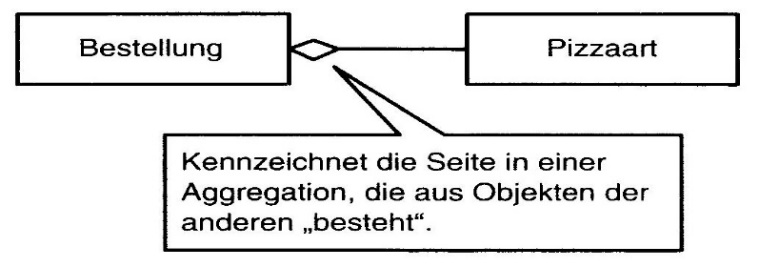
(Aus dem Buch Objektorientierte Analyse & Design: O’Reilly-Verlag, ISBN-Nr: 978-3-89721-495-89

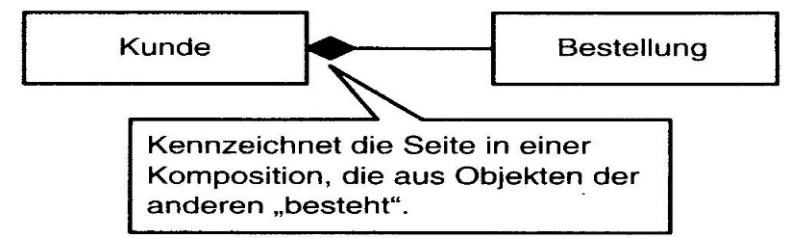


## Bidirektionale Beziehungen automatisch setzen

Da Bidirektionalität in Objektbeziehungen immer über zwei Verweise hergestellt werden muss, bedingt eine Änderung auf der einen Seite immer auch eine synchrone Änderung auf der anderen. Auch hier müssen Sie sich die Frage stellen, was z.B. mit Objekten passiert, wenn sie von ihrem Partner abgekoppelt werden oder der Partner gar verschwindet?

## Aggregation und Komposition: Spezialfälle einer Assoziation

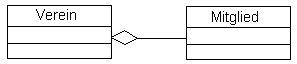
Wenn Sie genauer hinschauen, unterscheiden sich die Beziehungen zwischen Filialen und Personen sowie Bestellungen und Pizzaarten jedoch noch in einer Weise, die auch unabhängig von der Multiplizität ist: Filialen und Personen stehen quasi wirklich gleichberechtigt nebeneinander, Bestellungen hingegen enthalten Pizzaarten. Pizzaarten sind damit Bestellungen in der Beziehung untergeordnet. Pizzaarten konstituieren sozusagen eine Bestellung. Wo eine Assoziation die Form einer solchen Ganzes/Teile-Beziehung hat, spricht man von einer Aggregation.

Die Partner in einer Aggregation existieren immer noch unabhängig voneinander, die Child-Objekte haben jedoch die Rolle von Bestandteilen ihres Parent- bzw. ihrer Parent-Objekte. Aggregationen können uni- oder bidirektionale Beziehungen sein, setzen das Verhältnis vom Ganzen zu den Teilen gewöhnlich aber auf 1:n oder n:m.

In der Implementation besteht zwischen Assoziationen und Aggregationen kein Unterschied. Dieser ist eher logischer Art. Anders liegt der Fall jedoch, wenn bei einer Aggregation die Child-Objekte eben nicht unabhängig vom Parent existieren sollen. Dann spricht man von einer Komposition.

Insofern ist die Beziehung zwischen Kunden und Bestellungen keine Aggregation, sondern eine Komposition: Es gibt keine Bestellungen ohne Kunde und wenn ein Kunde gelöscht wird, sollten auch seine Bestellungen aus dem System verschwinden. Das hat Einfluss auf die Implementation des Beziehungsaufbaus zwischen den Objekten. Sie können zwar nicht verhindern, dass Instanzen der abhängigen Child-Klasse »einfach so« erzeugt werden, und Sie können auch nicht wirklich sicherstellen, dass bei Zerstörung eines Parent-Objekts alle Child-Objekte ebenfalls sicher zerstört werden, doch Sie sollten zumindest erzwingen, dass nicht beliebige Instanzen in eine Beziehung eingehängt werden. Das geschieht am besten über eine Fabrikmethode auf der Parent-Klasse.

**Aggregation (existenzunabhängige “hat“- oder “*ist-Teil-von“-Beziehung)***

Eine spezielle Form der Assoziation ist die Aggregation. Sie wird zur Modellierung genutzt, wenn die Beziehung zwischen den Objekten unterschiedlicher Klassen sehr eng ist und mit einer “ist-Teil-von“- Beziehung charakterisiert werden kann.

Ein Mitglied ist in diesem Sinne also ein Mitglied eines Vereins. Das Mitglied kann aber auch ohne Verein existieren, das heisst es ist existenziell unabhängig. Die Verbindung zwischen einem Verein und seinen Mitgliedern ist noch so eng, dass der Verein ohne Mitglieder in der Existenz bedroht ist und etwas mehr als eine einfache Assoziation vorliegt. Ein Mitglied kann zu mehreren Vereinen gehören. Wenn man einen Verein löscht, sollten die Mitglieder weiter bestehen, also nicht gelöscht werden.

Werden die Verbindung zwischen den Objekten noch loser, wie bei Vortragender und Zuhörer, dann sollte man eine Assoziation modellieren, denn hierbei handelt es ich um eine Beziehung die zwar wichtig sein kann, beide Seiten aber nicht in der Existenz bedroht.

**Komposition (existenzabhängige “hat“- oder “*ist-Teil-von“-Beziehung)***

Der Unterschied zwischen Aggregation und Komposition liegt darin, dass die enthaltenen Objekte existenziell abhängig sind vom Aggregations-Objekt, d.h., wird das Aggregations-Objekt zerstört, sind auch die enthaltenen Objekte nicht mehr existent.

Bei einer Aggregation wird die Verbindungslinie auf der Aggregat-Seite durch eine Raute ergänzt, bei der Komposition ist das Innere der Raute schwarz.

# 