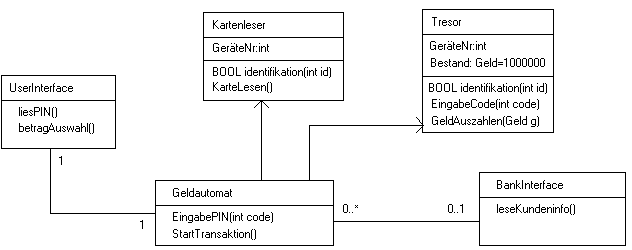
**Modul 226**

***Block 01***

***- Klassenbeziehungen I -***



**Inhaltsverzeichnis:**

[1 Repetition der theoretischen Grundlagen des Moduls 404 2](#_Toc69910488)

[2 Die lose Beziehung (use- & instanziiert) 3](#_Toc69910489)

[3 Die Assoziationsbeziehung 4](#_Toc69910490)

[3.1 Unidirektionale-Assoziationen 4](#_Toc69910491)

[3.2 Bidirektionale Assoziationen 5](#_Toc69910492)

[3.3 Gerichtete 1:n Beziehungen 5](#_Toc69910493)

[3.4 Bidirektionale 1:n Beziehungen 7](#_Toc69910494)

[3.5 Beziehungen des Typs n:m 8](#_Toc69910495)

[3.6 Ein kombiniertes Beispiel 8](#_Toc69910496)

[3.7 Assoziation, Aggregation, Komposition 9](#_Toc69910497)

[4 Übungen 11](#_Toc69910498)

[4.1 Aufgabe 1 (Fehler in Klassenbeziehungen erkennen) 11](#_Toc69910499)

[4.2 Aufgabe 2 (vom abstrakten Programmcode zum Klassendiagramm) 12](#_Toc69910500)

[4.3 Aufgabe 3 (vom realen Programmcode zum Klassendiagramm) 13](#_Toc69910501)

[4.4 Aufgabe 4 (Modellieren von Klassen und Bezieh. durch Textanalyse I) 14](#_Toc69910502)

[4.5 Aufgabe 5 (Modellieren von Klassen und Bezieh. durch Textanalyse II) 15](#_Toc69910503)

[4.6 Aufgabe 6 (Personen-Applikation v2.0) 17](#_Toc69910504)

[4.7 Aufgabe 7 (Personen-Applikation v3.0) 27](#_Toc69910505)

**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.

# Repetition der theoretischen Grundlagen des Moduls 404

**Aufgabe 1a**

Lernziel: Die Lernenden überprüfen ihren Kenntnisstand der theoretischen OOP Konzepte des M404

Form / Zeit: Partnerarbeit, 15’

Aufgabe: Untenstehend sind die wichtigsten Erkenntnisse des Modul404 Unterrichts aufgelistet. Gehen Sie diese Punkt für Punkt mit Ihrem Nachbarn durch. Erklären Sie diese einander und besprechen Sie, ob sie dasselbe darunter verstehen. Färben Sie die Punkte, die Ihnen geläufig und zu 100% klar sind mit grüner Farbe ein. Verwenden Sie die gelbe Farbe für die Aussagen, die Ihnen nur teilweise klar sind. Die rote Farbe verwenden Sie bei Aussagen, die Ihnen absolut unklar sind. Im Plenum erstellen wir danach eine Statistik der Situation und leiten den individuellen Nachschulungsbedarf ab.

Wir möchten die wichtigsten Feststellungen des bereits gelernten repetieren und an Beispielen festigen.

*01) Aus einer Klasse entstehen Objekte. Dieser Vorgang nennt man Instanziierung.*

*02) Ein Objekt nennt man auch die Instanz einer Klasse*

*03) Aus einer Klasse können beliebig viele Objekte erstellt werden.*

*04) Jedes Objekt basiert auf einer Klasse.*

*05) Wird eine Variable innerhalb einer Klasse, aber ausserhalb deren Methoden ohne die Schlüsselwörter const oder static deklariert, nennen wir Sie Membervariable (oder auch Eigenschaft, Instanzvariable oder Objektvariable)*

*06) Wird eine Variable innerhalb einer Klasse, aber ausserhalb deren Methoden mit einem der beiden Schlüsselwörter const oder static deklariert, nennen wir Sie Klassenvariable.*

*07) Gehört eine Funktion zu einer Klasse, nennen wir Sie Methode*

*08) Die Methoden einer Klasse können auf alle Membervariablen der Klasse zugreifen, egal ob diese Membervariablen public oder private sind.*

*09) Auf Membervariablen und Methoden mit der Sichtbarkeit public kann von ausserhalb der Klasse, via Referenz, zugegriffen werden*

*10) Auf Membervariablen und Methoden mit der Sichtbarkeit private kann von ausserhalb der Klasse, via Referenz, NICHT zugegriffen werden*

*11) Die Aufgabe des Softwareentwicklers ist, Klassen zu erstellen und zu beschreiben bei welchen Aktionen Objekte dieser Klassen erstellt werden sollen.*

*12) Während der Programmierung existiert kein einziges Objekt sondern nur Klassen.*

*13) Während der Programmausführung entstehen Instanzen (Objekte) aus den Klassen und interagieren miteinander.*

*14) Statische Eigenschaften werden beim Programmstart erstellt. Sie existieren genau einmal, ganz egal ob kein, ein oder hunderte von Objekten der Klasse erzeugt wurden.*

*15) Statische Methoden dürfen nur auf statische Eigenschaften zugreifen.*

*16) Auf statische Elemente kann man zugreifen, ohne erst eine Instanz (Objekt) erstellen zu müssen.*

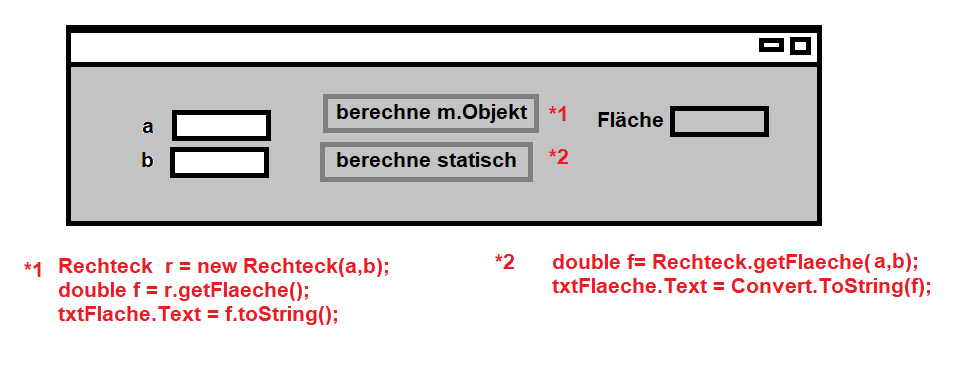
**Aufgabe 1b**

Lernziel: Die Lernenden überprüfen ihren Kenntnisstand der praktischen OOP Konzepte des M404

Form / Zeit: Einzelarbeit, 35’

Aufgabe: Erstellen sie mit c# oder Java ein Projekt, welches das unten ersichtliche Formular darstellt. Sie können zwei gebrochene Werte (z.B. 3.75 und 2.4) für die Rechteckseiten a und b eingeben und danach auf eine der beiden Befehlsschaltflächen klicken um die Fläche zu berechnen. Dies kann auf zwei Arten erfolgen (\*1) via Objekt oder (\*2) via statischer Methode.

Erstellen Sie den dafür nötigen Programmcode des Formulars und der Rechteck-Klasse.

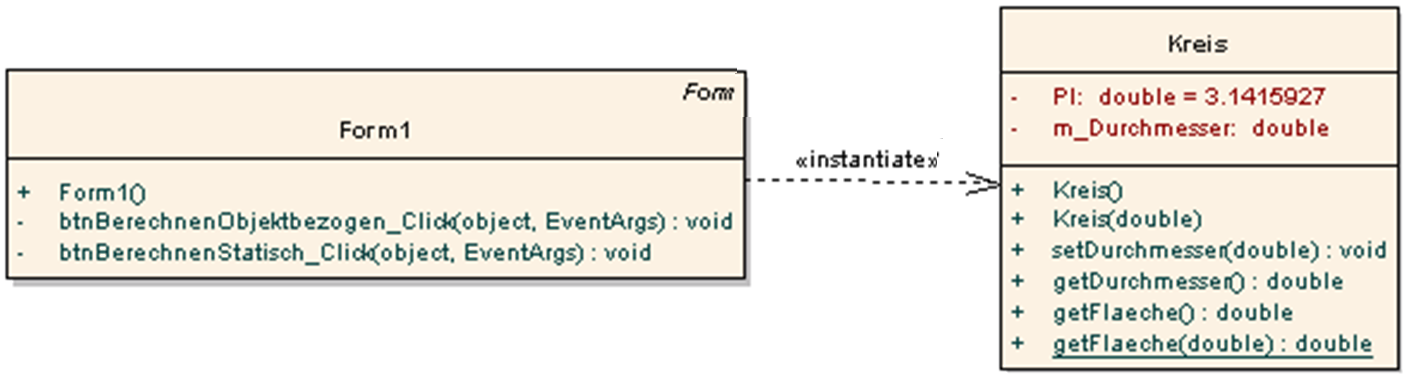


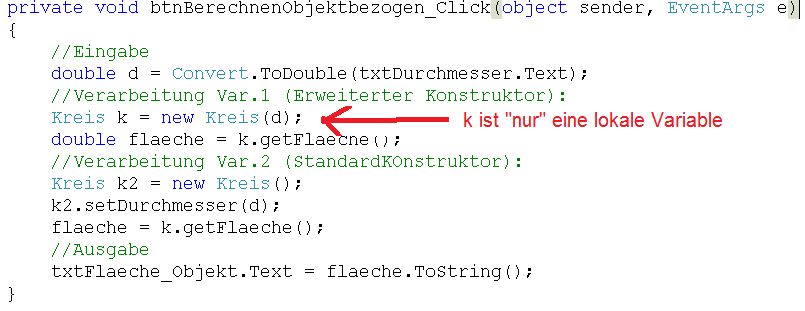
# Die lose Beziehung (use- & instanziiert)

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):**

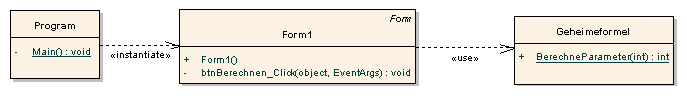
Wenn wir ein Objekt erstellen, welches ein anderes Objekt verwendet, ohne dessen Referenz als Membervariable zu speichern, handelt es sich um eine lose «inszanziiert» Beziehung.

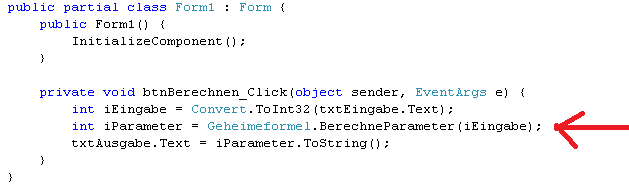




**Typ 2 «use» (Zugriff via Klassenname auf ein statisches Element einer anderen Klasse):**

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..\*.

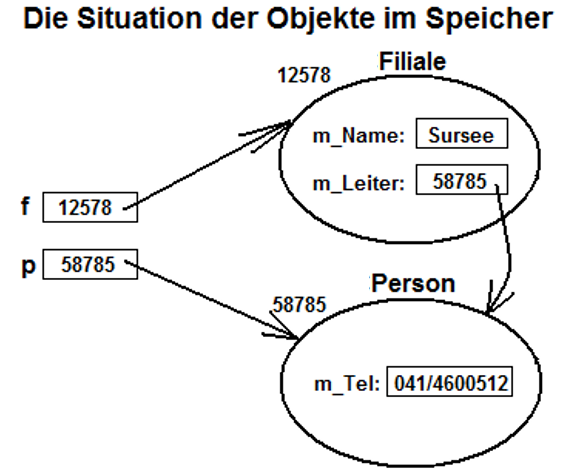
Die angegebene Multiplizität 4..18 steht für die Forderung, dass jedes Praktikum aus zumindest vier bis maximal 18 Studenten bestehen sollte. Ein Student kann an einer nicht näher spezifizierten Anzahl von Praktika teilnehmen.

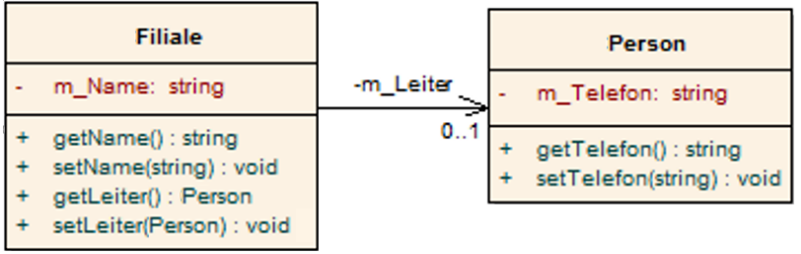
## Unidirektionale-Assoziationen

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;

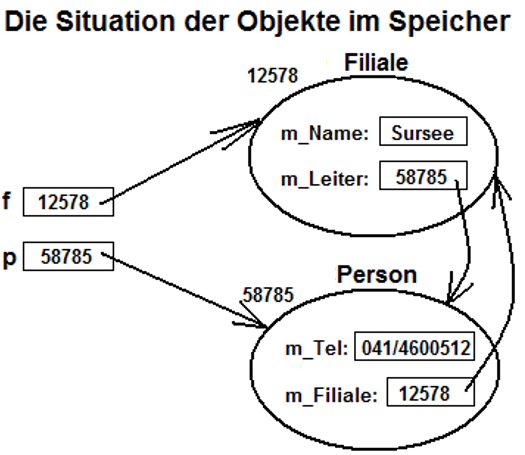
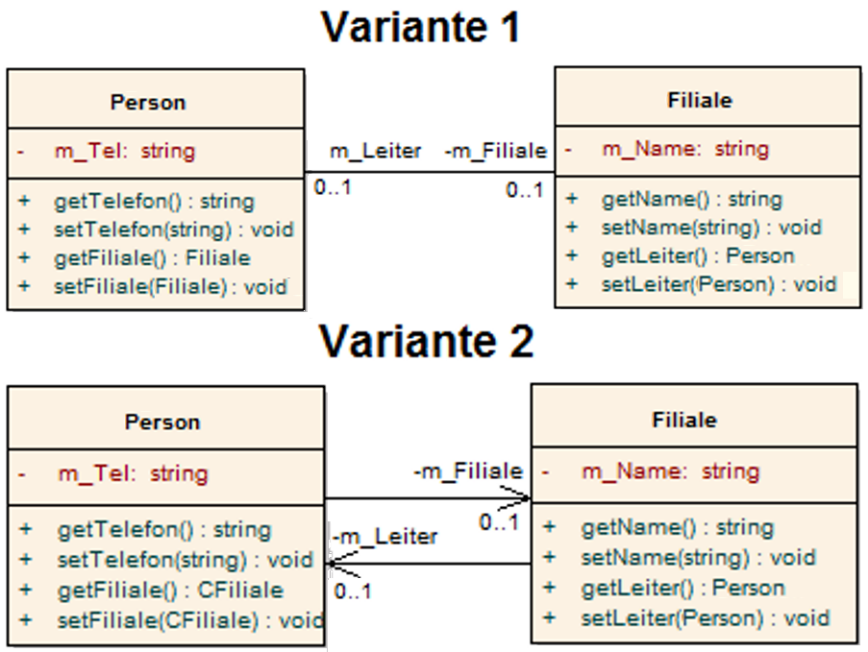
}

}

## 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;

}

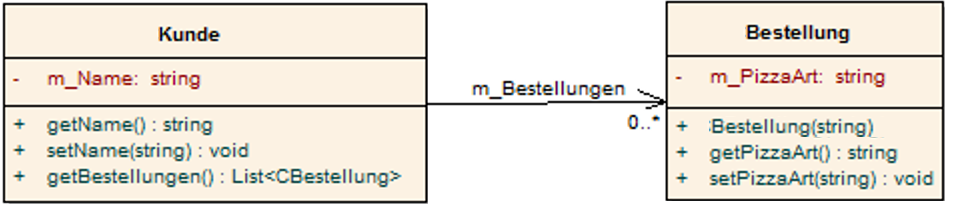
}

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){

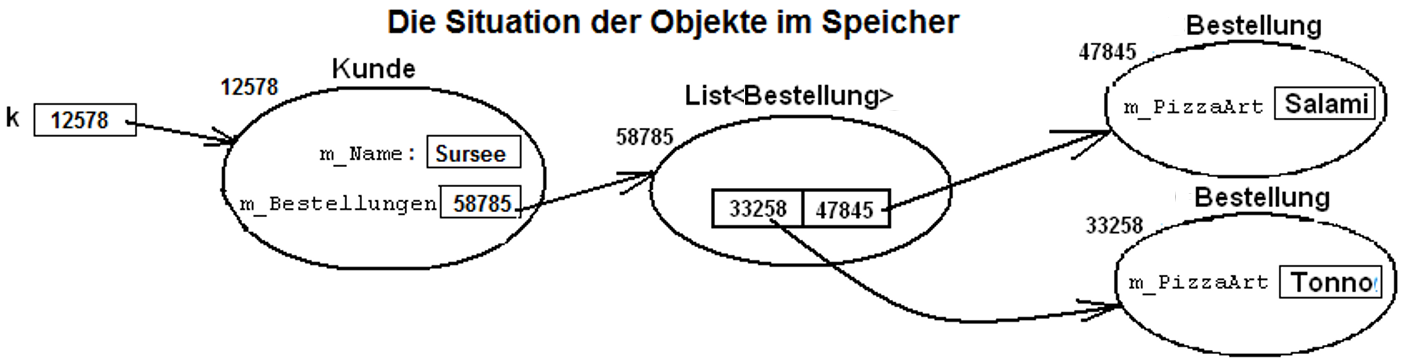
Kunde k = new Kunde();

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

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

}

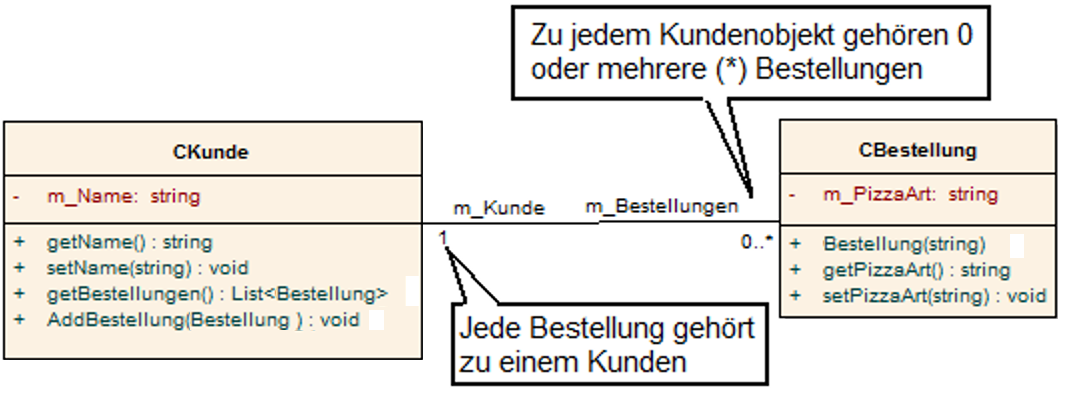
}



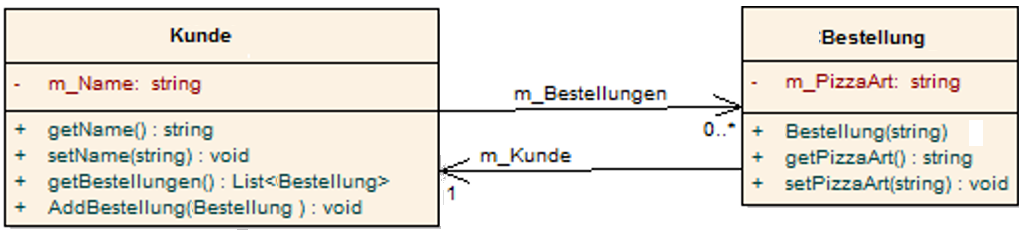
Das Feld m\_Bestellungen fasst alle Objekte der Child-Klasse (Bestellung) zusammen und unterstreicht damit den Listencharakter der Beziehung. Wenn sie die Zahl der angebundenen Child-Objekte sehr flexibel verändern wollen, ist ein statisches Datenfeld nicht die richtige Wahl, da diese eine feste Länge 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();

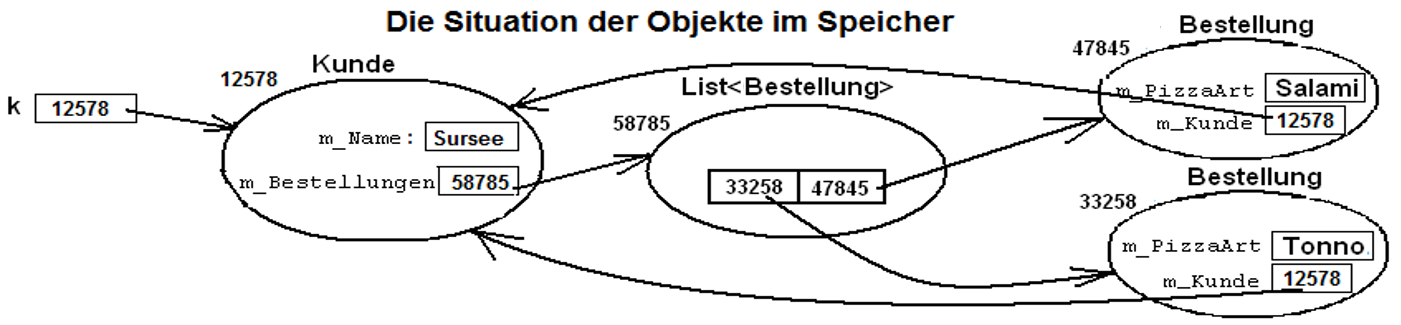
Bestellung b = new Bestellung("Tonno");

k.AddBestellung(b);

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

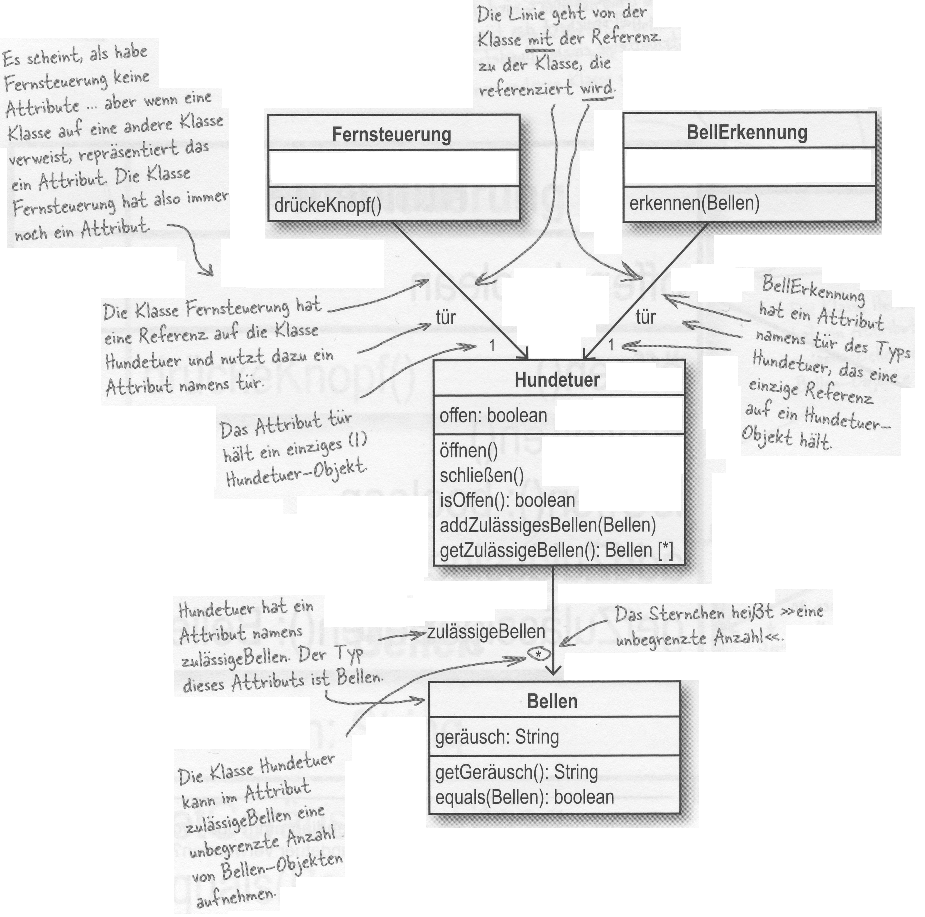
}

}

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

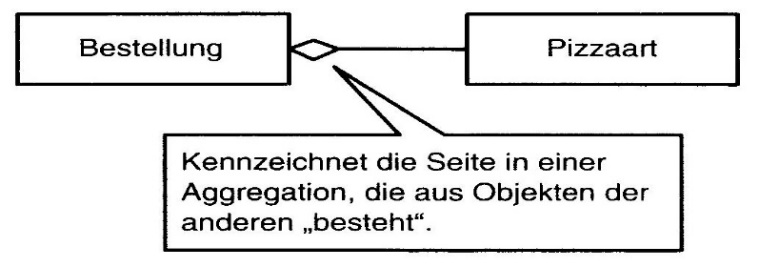
Wenn Sie genauer hinschauen, unterscheiden sich die Beziehungen zwischen Filialen und Personen sowie Bestellungen und Pizzaarten jedoch noch in einer Weise, die auch unabhängig von der Multiplizität ist: Filialen und Personen stehen quasi wirklich gleichberechtigt nebeneinander, Bestellungen hingegen 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.

Abbildung Aggregation //besteht aus

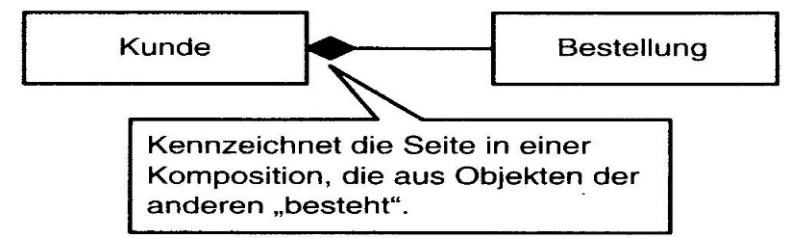
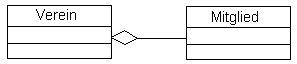
Die Partner in einer Aggregation existieren immer noch unabhängig voneinander, die Child-Objekte haben jedoch die Rolle von Bestandteilen ihres Parent- bzw. ihrer Parent-Objekte. Aggregationen können uni- oder bidirektionale Beziehungen sein, setzen das Verhältnis vom Ganzen zu den Teilen gewöhnlich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

Abbildung : Komposition ist //existenziell

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.

# Übungen

## Aufgabe 1 (Fehler in Klassenbeziehungen erkennen)

**Lernziel**: Beziehungen in Klassendiagrammen korrekt darstellen können

**Zeit**: 10’

**Aufgabe**: Gegeben sind die nachfolgenden Klassendiagramme. Beurteilen Sie, ob diese Diagramme richtig sind. Korrigieren Sie die fehlenden Diagramme.

|  |  |  |
| --- | --- | --- |
| Klassendiagramm | Richtig 🞎 falsch 🞎 | Richtiges Klassendiagramm |
|  | . |  |
| Klassendiagramm | Richtig🞎 falsch 🞎 | Richtiges Klassendiagramm |
|  |  |  |
| Klassendiagramm | Richtig🞎 falsch 🞎 | Richtiges Klassendiagramm |
|  | . |  |

## Aufgabe 2 (vom abstrakten Programmcode zum Klassendiagramm)

**Lernziel**: Aus gegebenem Programmcode ein Klassendiagramm erstellen.

**Zeit**: 12’

**Aufgabe**: Erstellen Sie das Klassendiagramm zur folgenden Programmcode-Situation.

public class **A**{

protected int m\_A;

public A(int a){

setA(a);

}

public int getA(){

return m\_A;

}

public void setA(int value){

m\_A = value;

}

public void TuWas(){

D.incCount();

}

}

public class **B**{

private int m\_B;

private List<D> m\_Ds = new List<D>();

public B(){

setB(0);

}

public int getB(){

return m\_B;

}

public void setB(int value){

m\_B = value;

}

}

public class **C** {

private B b = null;

public C(){

b = new B();

}

public void MachWas(){

D d = new D();

d.setD(10);

}

}

public class **D**{

private int m\_D;

private static int count = 0;

public int getD(){

return m\_D;

}

public void setD(int value){

m\_D = value;

}

public static void incCount(){

count++;

}

}

public class **Programm** {

static void Main(string[] args) {

B re = new B();

re.setB(2);

}

}

## Aufgabe 3 (vom realen Programmcode zum Klassendiagramm)

**Lernziel**: Aus Programmcode ein Klassendiagramm erstellen

**Zeit**: 20’

**Aufgabe:** a) Betrachten Sie den untenstehenden Programmcode und überlegen Sie sich, was das Programm ausgibt.

b) Zeichnen Sie für den gegebenen Programmcode das UML-Klassendiagramm inkl. den Beziehungen auf.

class **Program** {

static void Main(string[] args) {

Zylinder z = new Zylinder();

z.setHoehe(12);

z.setDurchmesser(5);

Console.WriteLine("Volumen = " + z.getVolumen());

Zylinder ik = new Zylinder(5,12);

Console.WriteLine("Volumen = " + ik.getVolumen());

Console.ReadLine();

}

}

class **Zylinder** {

private Kreis m\_Grundflaeche;

private int m\_Hoehe;

public Zylinder() {

m\_Grundflaeche = new Kreis(0);

setHoehe(0);

}

public Zylinder(int d, int h) {

m\_Grundflaeche = new Kreis(d);

setHoehe(h);

}

public int getHoehe() {

return m\_Hoehe;

}

public void setHoehe(int value) {

m\_Hoehe = value;

}

public int getDurchmesser() {

return m\_Grundflaeche.getDurchmesser();

}

public void setDurchmesser(int value) {

m\_Grundflaeche = new Kreis(value);

}

public double getVolumen() {

return m\_Grundflaeche.GetFlaeche() \* getHoehe();

}

public double getStandflaeche() {

return m\_Grundflaeche.GetFlaeche();

}

}

public class **Mathematik** {

public const double PI = 3.1415927;

public static int Quadrat (int Basis, int Exponent){

int res = 1;

for (int i = 1; i <= Exponent; i++)

res = res \* Basis;

return res;

}

public static double Addition(double z1, double z2) {

return z1 + z2;

}

}

public class **Kreis** {

private int m\_Durchmesser;

public Kreis() {

setDurchmesser(0);

}

public Kreis(int d) {

setDurchmesser(d);

}

private void setDurchmesser(int value) {

m\_Durchmesser = value;

}

public int getDurchmesser() {

return m\_Durchmesser;

}

public double GetFlaeche() {

return Mathematik.Quadrat(m\_Durchmesser, 2) \* Mathematik.PI / 4;

}

}

## Aufgabe 4 (Modellieren von Klassen und Bezieh. durch Textanalyse I)

**Lernziel**: Modellieren von Klassen und deren Beziehungen anhand von Textanalyse

**Zeit**: 10’

**Aufgabe**: Modellieren Sie folgende Problemstellungen als Klassendiagramme. Denken Sie sich für die Klassen geeignete Attribute (Membervariablen) aus. Nomen sind in der Regel Klassen und Verben sind Methoden der Klassen.

a) Mehrere Personen schliessen sich zu einer Fahrgemeinschaft zusammen

b) Ein Projektplan (z.B. Balkendiagramm) besteht aus mehreren Planungsschritten.

c) Ein Mitarbeiter tritt als Programmierer in ein Unternehmen ein. Nach ein paar Jahren wird er als Manager tätig und steigt später zum Geschäftsführer auf. Für Programmierer, Manager und Geschäftsführer sind unterschiedliche Eigenschaften festzuhalten Die Historie soll dabei abrufbar sein.

## Aufgabe 5 (Modellieren von Klassen und Bezieh. durch Textanalyse II)

**Lernziel**: Modellieren von Klassen und deren Beziehungen durch Textanalyse. Verwenden sie dabei bewusst noch keine Vererbung (wird erst im M226b verwendet). Erstellen Sie Ihr Diagramm mit dem CASE-Tool Enterprise Architect.

**Zeit**: 45’ (Einzelarbeit)

**Aufgabe**: Erstellen Sie anhand der folgenden Problembeschreibung ein grobes Klassen­diagramm.

**Text:** Eine Praxis mit mehreren Ärzten soll intern verwaltet werden. Für jeden Patienten sind Name, Adresse und Geburtsdatum zu speichern. Jeder Arzt vertritt bestimmte Fach­gebiete. Der Patient kann mehrere Ärzte dieser Praxis konsultieren. Für jede Be­handlung werden das Datum, die Diagnose und die erteilten Verordnungen fest­ge­halten. Jede Verordnung umfasst die Packungsgrösse, das Medikament und ggf. eine Vor­schrift für die Anwendung. Mehrere Behandlungen werden gemeinsam ab­ge­rechnet. Die Abrechnung enthält das Rechungsdatum und den Behandlungs­zeitraum sowie die einzelnen Abrechungspositionen. Jede Position besteht aus einer laufenden Nummer, der erbrachten Leistung, dem Abrechnungssatz und den Kosten.

**Ihre Lösung:**

**Lösung Schritt 1 (Textanalyse):**

|  |  |  |  |
| --- | --- | --- | --- |
| **⏹Klassen** | **⏹Attribute** | **⏹Beziehungen zu Klassen** | **⏹Methoden** |
| Praxis | Name |  |  |
| Arzt |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Ihre Lösung Step 2 (Klassendiagramm):**

## Aufgabe 6 (Personen-Applikation v2.0)

**Lernziel**: Klassen und Beziehungen in Code umsetzen und mit UML darstellen

**Form:** Einzelarbeit ohne Kommunikation**.**

**Zeit**: 135 Minuten

**Aufgabe**: Wir erstellen schrittweise eine Personenverwaltung. Erstellen Sie Schritt für Schritt die nachfolgend beschriebenen Aktionen und notieren Sie sich Fragen/ Unklarheiten. Sie müssen am Ende jedes Schrittes Ihre Applikation ohne Fehler kompilieren können.

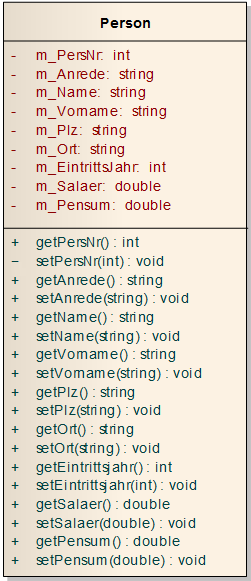
Personenverwaltung Schritt 1 (Projekt erstellen und vorbereiten):

Erstellen Sie in ein -Projekt mit der Bezeichnung “PersonenApp“.

Personenverwaltung Schritt 2 (Klasse Person erstellen und Membervariablen deklarieren):

Erstellen Sie eine neue Klasse mit der Bezeichnung Person. Ergänzen Sie die im nebenstehenden UML-Klassen­diagramm ersichtlichen Membervariablen.

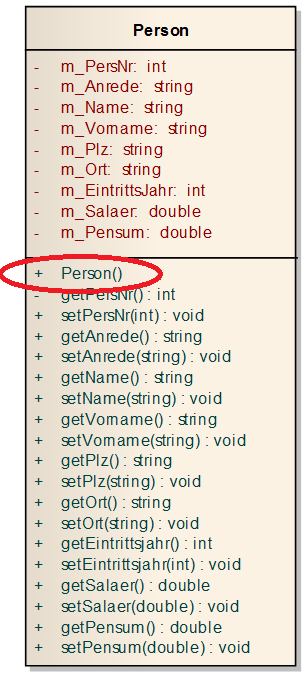
Achtung!! Das Minus (-) vor den Membervariablen bedeutet, dass diese Elemente die Sichtbarkeit privat besitzen.

Personenverwaltung Schritt 3 (get- / set-Methoden erstellen):

Erstellen Sie für alle Membervariablen get- und set-Methoden, damit die Inhalte der Membervariablen via Referenz gelesen, resp. verändert werden können.

Beachten Sie dabei aber folgendes:

* Das EintrittsJahr muss im Bereich 1975 bis (aktuelles Jahr) liegen. Wird einer dieser Grenzwerte verletzt, wird der entsprechende Grenzwert gespeichert und der Benutzer per Meldungsbox darauf hingewiesen.
* Das Salaer muss sich im Bereich 0.00 bis 99'999.95 bewegen, sonst wird gleich wie zuvor beschrieben verfahren. Erstellen Sie auch zwei Konstanten. Eine für das kleinste Eintrittsjahr und eine für das grösstmögliche Salär.sAnz
* Die PersNr darf von ausserhalb der Klasse nur gelesen werden (kein Schreibzugriff). Das Setzen von Werten ist nur via Konstruktor möglich (nächster Schritt).

Personenverwaltung Schritt 4 (Standardkonstruktor erstellen):

Erstellen Sie einen Standardkonstruktor, der die Eigenschaften auf folgende Werte setzt:

PersNr (integer) = -1

Anrede (string) = “Frau“

Namen (string) = “Neue Person“

Vornamen (string) = ““

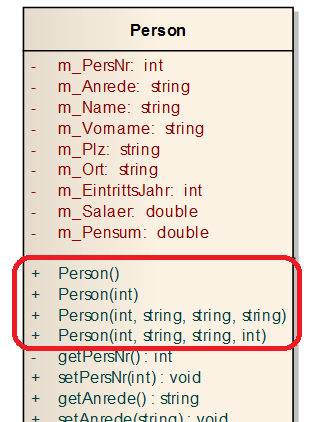
Plz (string) = “6000“

Ort (string) = “Luzern“

EintrittsJahr (integer) = aktuelles Jahr

Salaer (double) = “5000.00“ (entspricht dem Lohn bei 100%)

Pensum (double) = 100 (entspricht der Anstellung in %)



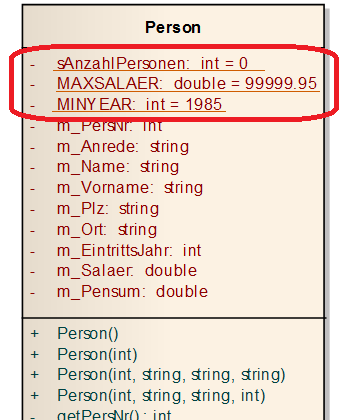
Personenverwaltung Schritt 5 (Konstruktoren mit Parameter erstellen):

Erstellen Sie folgende weitere Konstruktoren:

Dem Konstruktor 2 müssen Sie nur die PersNr übergeben, die übergebene PersNr wird natürlich in die Eigenschaft geschrieben.

Dem Konstruktor 3 müssen Sie die PersNr , die Anrede, den Namen und den Vornamen übergeben.

Dem Konstruktor 4 müssen Sie die PersNr , den Namen, den Vornamen und das Eintrittsjahr übergeben.

Personenverwaltung Schritt 6 (statische Eigenschaft):

Erweitern Sie Ihre Klasse um eine Klassenvariable (=statisch!!) vom Typ integer mit der Bezeichnung sAnzahlPersonen.

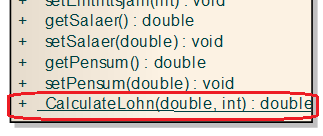
Der Wert, der in dieser Klassenvariablen steht, entspricht der Anzahl Personen-Objekte die bisher instanziiert wurden. Dazu setzen wir die Variable zu Beginn, bei der Deklaration auf Wert 0 (=noch kein Objekt erstellt) und erhöhen diesen Wert in jedem der Personen-Konstruktoren um eins).

Passen sie die Konstruktoren so an, dass die Personalnummer nicht mehr übergeben werden kann, sondern automatisch gesetzt wird.

Achtung!! Klassenvariablen werden in UML-Klassendiagrammen unterstrichen dargestellt. Konstanten haben einen zugeordneten Wert.

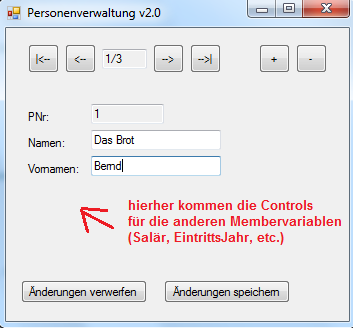
Personenverwaltung Schritt 7 (statische Methode):

Erstellen Sie eine statische Methode (eine Methode die auch ohne Instanz aufgerufen werden kann). Mit der Bezeichnung CalculateLohn().

CalculateLohn bekommt als Parameter zwei DoubleWerte (Salaer und Pensum). Das Salaer entspricht dem Lohn bei einer 100%igen Tätigkeit. CalculateLohn berechnet den effektiven Lohn und gibt diesen zurück.

z.B. double LohnEff = CPerson.CalculateLohn(5000.00, 85);

Erstellen Sie CalculateLohn noch in einer zweiten, nicht statischen Ausführung. Diese Methode enthält logischerweise keine Parameter und macht genau das gleiche, aber mit den aktuellen Objektwerten.

Personenverwaltung Schritt 8 (Ein Array aus Objekten):

Hauchen sie nun ihrer Personenverwaltung «Leben» ein.

Beim Anzeigen des Formulars sollen automatisch drei Personenobjekte erstellt und der erste Datensatz angezeigt werden.

Dazu müssen Sie auf den Klick der vier Navigationsbuttonereignisse und auf das Ereignis FormLoad (das Formular wurde fertig geladen) reagieren. Zusätzlich benötigen sie noch eine private Methode mit dem Namen FillForm().

Der Programmcode sollte nach der automatischen Erstellung der Ereignisse und dem manuellen Erstellen der Methode FillForm ungefähr folgendermassen aussehen:

*public partial class FormMain : Form {*

***//Membervariablen***

***//Konstruktoren***

*public FormMain() {*

*InitializeComponent();*

*}*

***//Ereignisse***

***private void OnFormLoad(object sender, EventArgs e) {***

***}***

***private void OnFirst(object sender, EventArgs e) {***

***}***

***private void OnPrevious(object sender, EventArgs e) {***

***}***

***private void OnNext(object sender, EventArgs e) {***

***}***

***private void OnLast(object sender, EventArgs e) {***

***}***

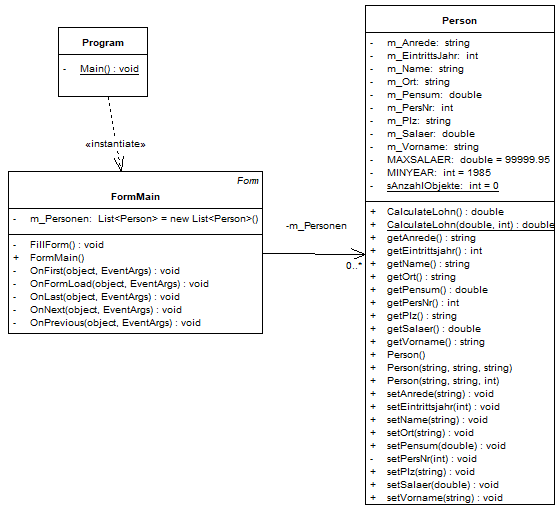
***//Methoden***

***private void FillForm() {***

***}***

*}*

Die Idee ist ja nun, dass beim Öffnen des Formulars drei Personenobjekte automatisch erstellt werden und das erste Objekt angezeigt werden soll. Um dies zu erreichen müssen wir eine gerichtete 1:n Assoziation zwischen der Klasse Formular und der Klasse Person erstellen.



Erstellen sie auch gleich bei der Formularklasse eine Membervariable, welche den Index der Person enthält, die aktuell auf dem Formular dargestellt wird (1 = erstes Personenobjekt, 2 = zweites…., etc.)

Der Programmcode sieht nun folgendermassen aus:

*public partial class FormMain : Form {*

***//Membervariablen***

**List<Person> m\_Personen = new List<Person>();**

**private int m\_Position = 0;**

***//Konstruktoren***

***…***

Passen sie nun das FormLoad-Ereigniss so an, dass automatisch der Personenobjekte erstellt und im Array gespeichert werden. Zeigen sie via FillForm Methode den aktuellen (den ersten) Datensatz an.

*public partial class FormMain : Form {*

private void OnFormLoad(object sender, EventArgs e) {

for (int i = 1; i <= 3; i++){

Person p = new Person();

p.setName("Neue Person (" + Convert.ToString(i) + ")");

m\_Personen.Add(p);

}

m\_Position = 1; *//Das erste Personenobjekt soll visualisiert werden*

FillForm();

}

private void FillForm() {

lblPosition.Text = Convert.ToString(m\_Position) + "/" + Convert.ToString(m\_Personen.Count);

if (m\_Personen.Count == 0)

return;

Person p = m\_Personen[m\_Position-1];

txtPNr.Text = Convert.ToString(p.getPersNr());

txtNamen.Text = p.getName();

txtVornamen.Text = p.getVorname();

//... usw

}

…

}

Nun ist es einfach, sich um die Navigationsmethoden zu kümmern. Der Code sollte selbsterklärend sein.

*public partial class FormMain : Form {*

private void OnFirst(object sender, EventArgs e) {

m\_Position = 1;

FillForm();

}

private void OnPrevious(object sender, EventArgs e) {

if (m\_Position > 1) {

m\_Position--;

FillForm();

}

}

private void OnNext(object sender, EventArgs e) {

if (m\_Position < m\_Personen.Count) {

m\_Position++;

FillForm();

}

}

private void OnLast(object sender, EventArgs e) {

if (m\_Personen.Count > 0) {

m\_Position = m\_Personen.Count;

FillForm();

}

}

}

Ermöglichen sie nebst der bereits realisierten Navigation auch das Speichern. Falls Sie <Änderung verwerfen> anklicken wird der Datensatz wieder so angezeigt, wie er zuletzt gespeichert wurde. Wenn sie «Änderungen speichern» wählen, wird der Inhalt des Formulars im Objekt (im Arbeitsspeicher also) übernommen. Dazu müssen sie auf die beiden Ereignisse reagieren:

*public partial class FormMain : Form {*

…

private void OnCancel(object sender, EventArgs e) {

FillForm();

}

private void OnSave(object sender, EventArgs e) {

Person p = m\_Personen[m\_Position-1];

p.setName(txtNamen.Text);

p.setVorname(txtVornamen.Text);

…

}

}

Personenverwaltung Schritt 9a (bestehende Personen löschen):

Erweitern Sie Ihre Personenverwaltung um die Funktionalität der Befehlsschaltfläche <->, mit der bestehende Personenobjekte gelöscht werden können. Wenn Sie <-> anklicken wird die aktuelle Person (jene die im Hauptfenster dargestellt wird) gelöscht.

Erstellen sie zuerst das entsprechende Ereignis:

*public partial class FormMain : Form {*

…

private void OnDelete(object sender, EventArgs e) {

}

…

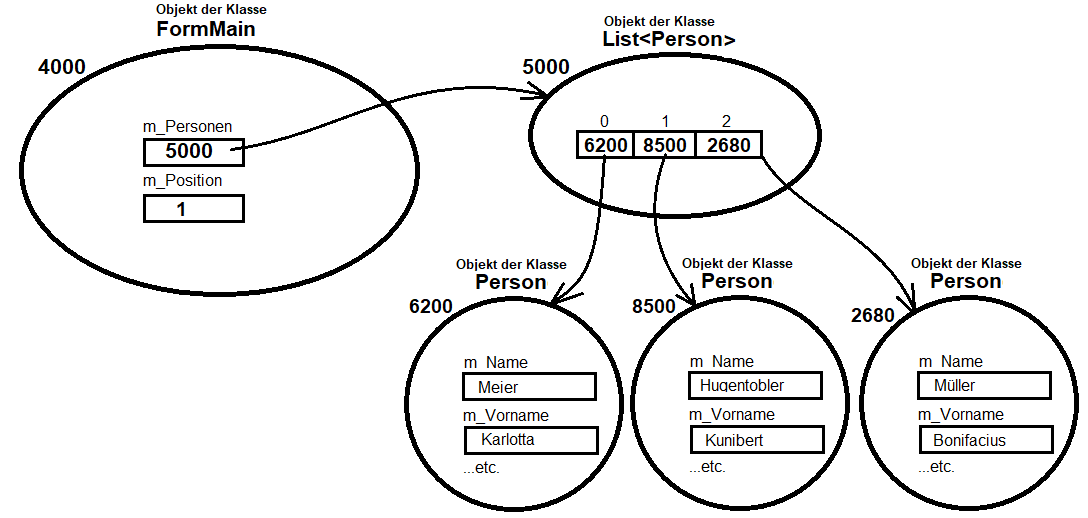
}

Das Löschen kann mit einer der beiden dafür vorgesehenen Methoden der List-Klasse erfolgen:

* RemoveAt (int index)
* Remove (object adresse)

Wenn wir also im untenstehenden Fall das Personenobjekt Kunibert Hugentobler löschen möchten haben wir zwei Möglichkeiten:

* m\_Personen.RemoveAt(1); // Die Zahl 2 enstpricht dem Index (bei 0 beginnend)
* m\_Personen.Remove(8500); // Die Zahl 7500 entspricht der Objektadresse des Personenobjektes



Damit wird auch der Programmcode des Klickereignisses nachvollziehbar und verständlich:

*public partial class FormMain : Form {*

…

private void OnDelete(object sender, EventArgs e) {

m\_Personen.RemoveAt(m\_Position-1) ; // Die aktuelle Person (an der Stelle m\_Position) löschen

if (m\_Position > m\_Personen.Count) // Falls das die letzte Person im Array war, die gelöscht wurde,…

m\_Position--; // soll der Vorgänger die neue aktuelle Person sein

FillForm();

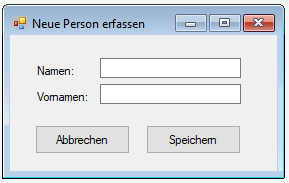
}

…

}

Personenverwaltung Schritt 9b (neue Personen hinzufügen):

Erweitern Sie Ihre Personenverwaltung um die Funktionalität der Befehlsschaltfläche <+>, mit der neue Personen erfasst werden können. Wenn Sie <+> anklicken, soll ein Formularobjekt der Klasse FormNeu instanziiert und modal angezeigt werden. Das FormNeu-Objekt enthält nur zwei Textfelder (Namen & Vornamen) und zwei Befehlsschaltflächen <Abbrechen> und <Speichern>.

Fügen Sie ihrem Projekt eine zweite Formularklasse hinzu. Klicken sie dazu im Projektmappenexplorer die rechte Maustaste auf dem Projekt und wählen sie im Kontextmenü den Eintrag Hinzufügen🡪Neues Element» aus. In der Auflistung möglicher Elemente wählen Sie den Eintrag WindowsForm. Wählen Sie als Klassenname «FormNeu».

Designen sie die Formularklasse

wie nebenstehend ersichtlich.

Erstellen sie beim Hauptformular das Ereignis für den Klick auf den Button <+> das sie OnAddPerson nennen. Instanziieren sie in diesem Ereignis ein Objekt der Klasse FormNeu und zeigen sie das Fenster als modales Fenster an (mit der Methode showDialog() nicht mit show() !! ).

*public partial class FormMain : Form {*

…

private void OnAddPerson(object sender, EventArgs e) {

FormNeu f = new FormNeu();

f.ShowDialog(); // das Fenster wird modal angezeigt. Das heisst die Codeausführung fährt an dieser Stelle erst dann

// weiter, wenn das «Neue Person erfassen»-Fenster geschlossen wurde.

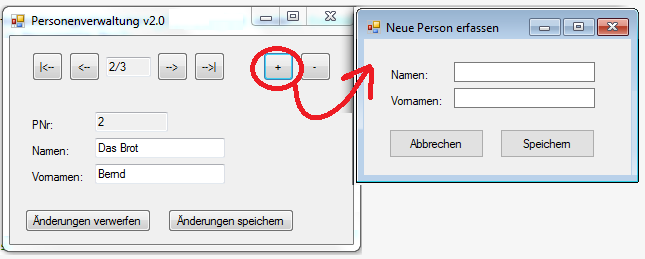
// Geschlossen heisst: nicht mehr auf dem Bildschirm, aber noch im Arbeitsspeicher zugreifbar

}

…

}

Der folgende Sachverhalt sollte nun bereits funktionieren. Testen sie das Verhalten des neuen Fensters. Sie werden bemerken, dass solange dieses modale Fenster angezeigt wird, kein anderes Fenster der Applikation selektiert werden kann. Solch modale Fenster sind typisch für Eingaben.



Nach dem Bestätigen der Befehlsschaltflächen <Speichern> oder <Abbrechen> soll das modale Fenster geschlossen werden. Wir möchten an der aufrufenden Stelle im Hauptformular später unterscheiden können, ob das Erfassenfenster mit Speichern oder Abbrechen verlassen wurde. Darum setzen wir die Membervariable Dialogresult auf den entsprechenden Wert (Enumerator).

*public partial class FormNeu : Form {*

*public FormNeu() {*

*InitializeComponent();*

*}*

private void OnAbort(object sender, EventArgs e) {

DialogResult = DialogResult.Abort;

}

private void OnSave(object sender, EventArgs e) {

DialogResult = DialogResult.OK;

}

}

Ergänzen sie nun den folgenden Programmcode beim Formular FormNeu:

*public partial class FormNeu : Form {*

***//Membervariablen***

**private Person m\_NeuePerson = null;**

*public FormNeu() {*

*InitializeComponent();*

*}*

*private void OnAbort(object sender, EventArgs e) {*

*DialogResult = DialogResult.Abort;*

*}*

*private void OnSave(object sender, EventArgs e) {*

**m\_NeuePerson = new Person();**

**m\_NeuePerson.setName(txtNamen.Text);**

**m\_NeuePerson.setVorname(txtVornamen.Text);**

*DialogResult = DialogResult.OK;*

*}*

**public Person getNeuePerson() {** // benötigen wir um via Referenz im Hauptformular die Adresse des

// neuen Personenobjektes zu bekommen

**return m\_NeuePerson;**

**}**

}

Ergänzen sie nun den folgenden Programmcode beim Formular Hauptformular:

*public partial class FormMain : Form {*

…

private void OnAddPerson(object sender, EventArgs e) {

FormNeu f = new FormNeu();

f.ShowDialog();

**if (f.DialogResult == System.Windows.Forms.DialogResult.OK) {**

**Person p = f.getNeuePerson();**

**m\_Personen.Add(p);**

**OnLast(this, null);** //Das Ereignis OnLast (-->|) soll ausgeführt werden

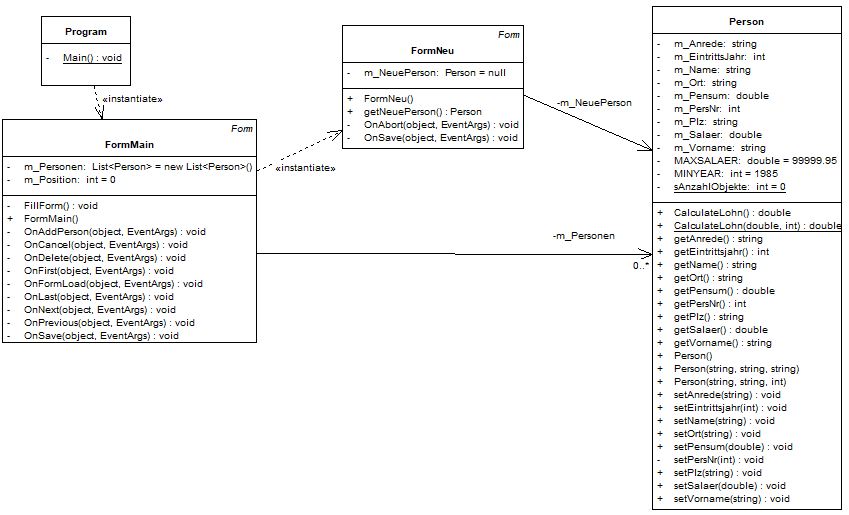
**}**

}

…

}

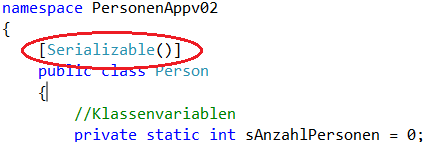
Per UML sieht die Situation nun so aus:



Personenverwaltung Schritt 10a (Serialisierung der Objekte in eine Datei):

Erweitern Sie Ihre Personenverwaltung so dass die Personeninformationen nach der Beendigung in einer Datei gespeichert werden (man spricht von persistenter Speicherung).

Wir erledigen diese Aufgabe mit Hilfe der Serialisierung. Die Serialisierung erledigt die Umwandlung der Membervariablen von Objekten in eine Folge von hintereinander liegenden Bits (001010100100….). Die meisten von .Net bereitgestellten Klassen unterstützen diese Serialisierung bereits. Auch die dynamische Arrayklasse (List), die wir verwendet haben unterstützt die Serialisierung bereits. Das heisst, die dynamische Arrayklasse (List) serialisiert alle Objekte welche von ihr referenziert werden. Dies gelingt natürlich nur bei Objekten die auch die Serialisierung unterstützen, andernfalls würde eine Exception erfolgen. Unsere Personenobjekte, die von der Klasse Person instanziiert wurden, unterstützen diese Serialisierung noch nicht. Dies ändern wir, indem wir bei der Klassendefinition den Befehl [Serializable()] eingeben. Bei der Kompilierung werden dann die entsprechenden Methoden für die Serialisierung und Deserialisierung automatisch hinzugefügt.



Ergänzen Sie danach den Programmcode der Klasse HauptFormular um das Ereignis FormClosed. Testen Sie mit dem Dateiexplorer ob die Serialisierung erfolgreich war. Sie benötigen im Hauptformular zwei neue “Usings“:

namespace PersonenAppv02 {

public partial class **Hauptformular**: Form {

**…**

private void OnFormClosed(object sender, FormClosedEventArgs e) {

// Dieses Ereignis wird aufgerufen, wenn das Formular geschlossen wird (das Objekt ist allerdings immer

// noch im Arbeitsspeicher). Es erstellt eine Verbindung zu einer Datei (einen Stream) und speichert

// über das BinaryFormatter-Objekt die ganze dynamische Liste m\_AllePersonen in der Datei.

string sPfad = Application.StartupPath + "\\MyObject.pd7";

if (DialogResult.No == MessageBox.Show("Möchten Sie die Daten im Applikationsverzeichnis speichern?", "Speichern", Mes

sageBoxButtons.YesNo, MessageBoxIcon.Question))

{

SaveFileDialog sfd = new SaveFileDialog();

sfd.Filter = "pd7 files (\*.pd7)|\*.pd7|All files (\*.\*)|\*";

sfd.DefaultExt = "pd7";

if (DialogResult.OK == sfd.ShowDialog())

sPfad = sfd.FileName;

else

return;

}

**FileStream myStream = new FileStream(sPfad, FileMode.Create);**

**BinaryFormatter binFormatter = new BinaryFormatter();**

**binFormatter.Serialize(myStream, m\_Personen);**

**myStream.Close();**

}

}

}

Personenverwaltung Schritt 10b (Deserialisierung aus einer Datei):

Beim Programmstart werden im Konstruktor der Hauptformularklasse zurzeit immer drei neue Personenobjekte erstellt. Wir möchten dies nun so ändern, dass die Anwender wählen können, ob die drei Grundobjekte erstellt oder die Daten aus einer Datei eingelesen werden sollen.

Das heisst, wir fragen den Benutzer beim Starten der Applikation, ob er Daten aus einer Datei laden möchte. Falls Nein, erstellen wir wie bis anhin die drei Grundobjekte. Falls der Benutzer <Ja> wählt, fragen wir ihn mit einer zweiten Frage, ob er die Daten aus dem Applikationsverzeichnis laden möchte. Falls Ja, wird die Datei MyObject.pd7 aus dem Applikationsverzeichnis gelesen, falls Nein wir der Benutzer mit einem OpenFileDialog zur Wahl des Orders aufgefordert. Die Informationen der so gewählten Datei werden danach in Objekte deserialisiert (von hintereinander liegenden Bits (001010100100….) in Objekte.

Ändern Sie die OnFormLoad-MEthode wie folgt und ergänzen Sie die untenstehende Methode:

*public partial class FormMain : Form {*

private void OnFormLoad(object sender, EventArgs e) {

~~for (int i = 1; i <= 3; i++){~~

~~Person p = new Person();~~

~~p.setName("Neue Person (" + Convert.ToString(i) + ")");~~

~~m\_Personen.Add(p);~~

~~}~~

**Einlesen();**

m\_Position = 1;

FillForm();

}

**private void Einlesen () {**

if (DialogResult.No == MessageBox.Show("Möchten Sie die Daten aus einer Datei laden?", "Laden", MessageBoxBut

tons.YesNo, MessageBoxIcon.Question)) {

//Objekte nicht aus einer Datei laden sondern einfach drei Standardobjekte erstellen

for (int i = 1; i <= 3; i++) {

Person p = new Person();

p.setName("Neue Person (" + Convert.ToString(i) + ")");

m\_Personen.Add(p);

}

}

else {

//Objekte aus einer Datei laden

string sPfad = Application.StartupPath + "\\MyObject.pd7";

if (DialogResult.No == MessageBox.Show("Möchten Sie die Datei im Applikationsverzeichnis lesen?", "Informationen einlesen",

MessageBoxButtons.YesNo, MessageBoxIcon.Question)) {

OpenFileDialog ofd = new OpenFileDialog();

ofd.Filter = "pd7 files (\*.pd7)|\*.pd7|All files (\*.\*)|\*";

if (DialogResult.OK == ofd.ShowDialog())

sPfad = ofd.FileName;

else

return;

}

try{

**FileStream myStream = new FileStream(sPfad, FileMode.Open);**

**BinaryFormatter binFormatter = new BinaryFormatter();**

**m\_Personen = (List<Person>)binFormatter.Deserialize(myStream);**

**myStream.Close();**

}

catch (Exception ex){

//Falls ein Fehler passiert, Fehlermeldung ausgeben

MessageBox.Show(ex.Message);

}

}

}

}

Personenverwaltung Schritt 12 (Architektur mit UML-Klassendiagramm visualisieren):

Erstellen Sie mit der Software Enterprise Architect ein UML-Klassendiagramm, das alle Beziehungen anzeigt. Ergänzen Sie die losen Beziehungen.

Personenverwaltung Schritt 13 (Verbesserung der Details, Ergänzung, Fehlerbehandlung):

Unsere Applikation läuft noch nicht sehr stabil und hat einiges an Verbesserungspotenzial.

**Erweiterungsoption für “Schnelle“:**

Der Schritt 13 ist optional und dienen als zeitlicher Puffer. Das heisst, Lernende die sehr schnell vorwärtsgekommen sind, können sich damit ausführlicher befassen. Wenn Sie sehr knapp in der Zeit sind, lassen Sie diesen Schritt komplett weg.

Vorschlag 01:

Bearbeiten sie alle relevanten Codeabschnitte so, dass keine Abstürze mehr passieren können.

Vorschlag 02:

Es stört zum Beispiel, dass ich beim Beenden immer einen Pfad mitteilen muss. Ändern Sie die Applikation so, dass beim Schliessen des Hauptfensters nur noch nach dem Pfad gefragt wird, wenn die Daten ursprünglich nicht aus einer Datei geladen wurden (wenn quasi die drei Grundobjekte erstellt wurden). In allen andern Fällen sollen die Informationen wieder in der Datei gespeichert werden, aus welcher sie geladen wurden.

Vorschlag 03:

In der Regel arbeite ich meist immer mit einer bestimmten Datei. Ändern Sie die Applikation so, dass automatisch und ohne Rückfrage, die zuletzt geöffnete Datei wieder verwendet wird. Sollte diese Datei nicht mehr existieren, wird der Benutzer darauf hingewiesen und wie ursprünglich verfahren (Das heisst, wir fragen den Benutzer beim Starten der Applikation, ob er Daten aus einer Datei laden möchte….) .

Vorschlag 04:

Ergänzen Sie die Applikation so, dass pro Person auch eine E-Mailadresse erfasst werden kann.

***Achtung!! Bei einer Änderung der Membervariablen können natürlich Dateien mit dem alten Aufbau nicht mehr deserialisiert werden.***

## Aufgabe 7 (Personen-Applikation v3.0)

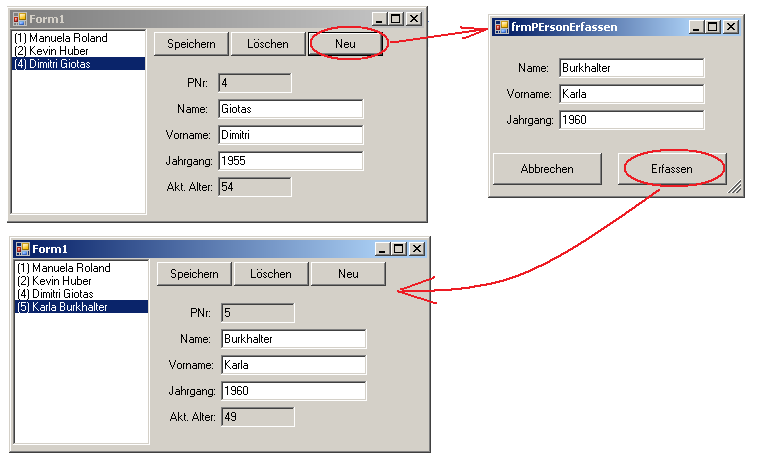
**Lernziel**: Klassen und Beziehungen in Code umsetzen und mit UML darstellen

**Form:** Einzelarbeit ohne Kommunikation**.**

**Zeit**: 135 Minuten

**Aufgabe**: Erstellen Sie, diesmal mit weniger Anleitung, eine neue Art Personenanwendung. Die Applikation soll nach dem folgenden Muster funktionieren.

Beschreibung:

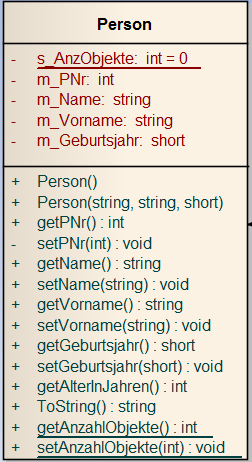


**Schritt 1** (Projekt erstellen und vorbereiten):

Erstellen Sie in eine C#-Windows-App-Projekt mit der Bezeichnung “PersonenAppV3“. Die App enthält wie oben ersichtlich, zwei Formulare: Das Hauptformular und das Erfassungsformular.

**Schritt 2** (GUI fertigstellen):

Die Lehrperson gibt ihnen wichtige Tipps zum Layout des Hauptformulars (Stichwort: Panels, Docking und ListBox). Schauen Sie zu und passen Sie danach Ihr Projekt entsprechend an. Die Controls sollen wie oben dargestellt erscheinen.



**Schritt 3** (Klasse Person erstellen):

Erstellen Sie nach folgendem UML-Klassendiagramm die Klasse Person mit allen Elementen. Achten Sie dabei auf folgendes:

Die Personalnummer wird im Konstruktor automatisch gesetzt. Von ausserhalb der Klasse kann die Personalnummer nur gelesen, nicht aber verändert werden.

Die unterstrichenen Elemente sind statisch.

Kapseln Sie alle Member- und Klassenvariablen durch get-/set-Methoden.

* getAnzahlObjekte gibt eine Kopie der statischen Variablen s\_AnzObjekte zurück
* setAnzahlObjekte schreibt den als Parameter übergebenen Wert in die statische Variable s\_AnzObjekte

Die set-Methode des Geburtsjahrs ist etwas komplexer: (siehe auch Tipp)

* Wenn das eingegebene Jahr 2-stellig ist und kleiner als das aktuelle Jahr (z.B. 07), dann wird automatisch das aktuelle Jahrhundert dazugerechnet (2007).
* Wenn das eingegebene Jahr 2-stellig ist und grösser als das aktuelle zweistellige Jahr (z.B. 88), dann wird automatisch das letzte Jahrhundert dazugerechnet (1988).
* Wenn die Eingabe grösser ist als das aktuelle Jahr, wird automatisch das aktuelle Jahr gesetzt (Bsp. Eingabe : 2018 🡪 2009)

Die getAlterInJahren() Methode berechnet wie alt die Person am aktuellen Tag ist. (siehe Tipp)

Die Methode ToString können Sie im Moment beiseitelassen. Die wird später erstellt.

Tipp: Das aktuelle Jahr bekommen Sie durch: DateTime.Now.Year

**Schritt 4** (Die Befehlsschaltfläche <Neu> implementieren)

Durch einen Klick auf die Befehlsschaltfläche <NEU> soll ein Objekt der frmPersonenErfassen-Fenster-Klasse modal angezeigt werden (Tipp: showDialog()-Methode). Wenn im besagten frmPersonenErfassen-Fenster Daten eingegeben werden und danach auf <Erfassen> geklickt wird, soll als Membervariable hinter dem frmPersonenErfassen-Fenster automatisch ein Personenobjekt erstellt werden, dessen Membervariablen - Inhalte aus den Textfeldern übernommen wird.

Das aufrufende Fenster (das Hauptformular) kann danach mittels getNeuePerson() Methode die Adresse des neuen Objektes bekommen und dieses neue Objekt im dynamischen Datenfeld (Array) speichern.

Lösen Sie diese Aufgabe wie bei der Aufgabe des vorherigen Kapitels. Holen Sie sich dort allenfalls Tipps.

**Schritt 5** (Die neu erfasste Person in der ListBox visualisieren)

Derzeit können Sie nur via Debug-Haltepunkt überprüfen, ob die Erfassung von Personen auch wirklich funktioniert. Auf dem Hauptfenster wird noch nichts angezeigt. Das möchten wir nun ändern. Vervollständigen Sie die Methode FuelleDialog wie nachfolgend beschrieben und überprüfen Sie das Ergebnis.

namespace PersonenAppv03 {

public partial class frmHaupt: Form {

…

private void FuelleDialog() {

//Zeigt den Inhalt des Arrays m\_AllePersonen in der Listbox an

}

}

Sie merken, dass pro Objekt ein Eintrag ausgegeben wird. Allerdings ist dieser Eintrag ziemlich nichtssagend, denn es handelt sich um den Klassennamen des erfassten Objektes.

Was passiert hier?

Wenn Sie ein Objekt der Klasse Person in eine Zeichenkette (String) umwandeln, wird automatisch der Klassenname ausgegeben.

Bertrachten wir das an einem Beispiel:

**Auto a = new Auto(123, “Mazda“, “MX-5“) ;**

**string s = Convert.ToString(a) ;**

Der Inhalt des Strings s ist nach der Ausführung des Programmcodes “Auto“. Warum? Weil das die oberste aller Klassen (sie heisst Object) standardmässig macht. Wenn wir das nicht möchten, müssen wir die ToString-Methode überschreiben.

Ergänzen Sie den Programmcode Ihrer Person Klasse um die folgende Methode und betrachten Sie danach das Verhalten Ihrer Applikation. Nun sollten in der ListBox des Hauptfensters Einträge in der Form “(22) Bernd Das Brot“ erscheinen:

namespace PersonenAppv03 {

public class Person {

…

public override string ToString() {

//Diese Funktion wird aufgerufen, wenn ein Peronenobjekt

//als Ganzes in eine Zeichenkette umgewandelt wird

return "(" + m\_PNr + ") " + m\_Vorname + " " + m\_Name;

}

}

}

**Schritt 6** (Die Details der in der Listbox ausgewählten Person auf der rechten Seite anzeigen)

Reagieren Sie auf das SelectedIndexChanged-Ereignis des ListBox-Objektes. Fragen Sie in dem Ereignis den Inhalt des SelectedItem -Property (das selektierte Objekt in der ListBox) ab und speichern Sie die Adresse des Objektes in einer lokalen Variable (Achtung casting ist nötig, siehe unten).

namespace PersonenAppv03

{

public partial class frmHaupt: Form

{

…

private void listBoxAllePersonen\_SelectedIndexChanged(object sender, EventArgs e) {

Person p = (Person)listBoxAllePersonen.SelectedItem;

…

}

}

}

**Schritt 7** (Die Befehlsschaltfläche <Speichern> implementieren)

Reagieren Sie auf das Klick-Ereignis der Befehlsschaltfläche <Speichern>. Fragen Sie in dem Ereignis den Inhalt des SelectedItem-Property (das selektierte Objekt in der ListBox) ab und speichern Sie die Adresse des Objektes in einer lokalen Referenz-Variablen (Achtung casting ist nötig, siehe unten). Wenn eine Person selektiert ist (Adresse nicht gleich null), lesen Sie alle Informationen aus den Textfeldern und schreiben Sie via set-Methoden in das selektierte Objekt.

Am Ende der Methode können Sie veranlassen, dass die ListBox neu mit dem Inhalt des Arrays gefüllt wird. Dazu müssen Sie Methode FuelleDialog() aufrufen. Soll die zuletzt ausgewählte Person wieder selektiert sein, können Sie die Objektadresse als SelectedItem festlegen (setzen).

private void cmdSpeichern\_Click(object sender, EventArgs e) {

Person p = (Person)listBoxAllePersonen.SelectedItem;

if (p == null)

return;

**???**

FuelleDialog();

listBoxAllePersonen.SelectedItem = p;

}

**Schritt 8** (Die Befehlsschaltfläche <Löschen> implementieren)

Reagieren Sie auf das Klick-Ereignis der Befehlsschaltfläche <Speichern>. Fragen Sie in dem Ereignis den Inhalt des SelectedItem-Property (das selektierte Objekt in der ListBox) ab und speichern Sie die Adresse des Objektes in einer lokalen Variable (Achtung casting ist nötig, siehe unten). Wenn eine Person selektiert ist (Adresse nicht gleich null), soll dieses Objekt aus den beiden Arrays (AllePersonen und listBoxAllePersonen.Items ) gelöscht werden.

private void cmdLoeschen\_Click(object sender, EventArgs e) {

Person p = (Person)listBoxAllePersonen.SelectedItem;

if (p == null)

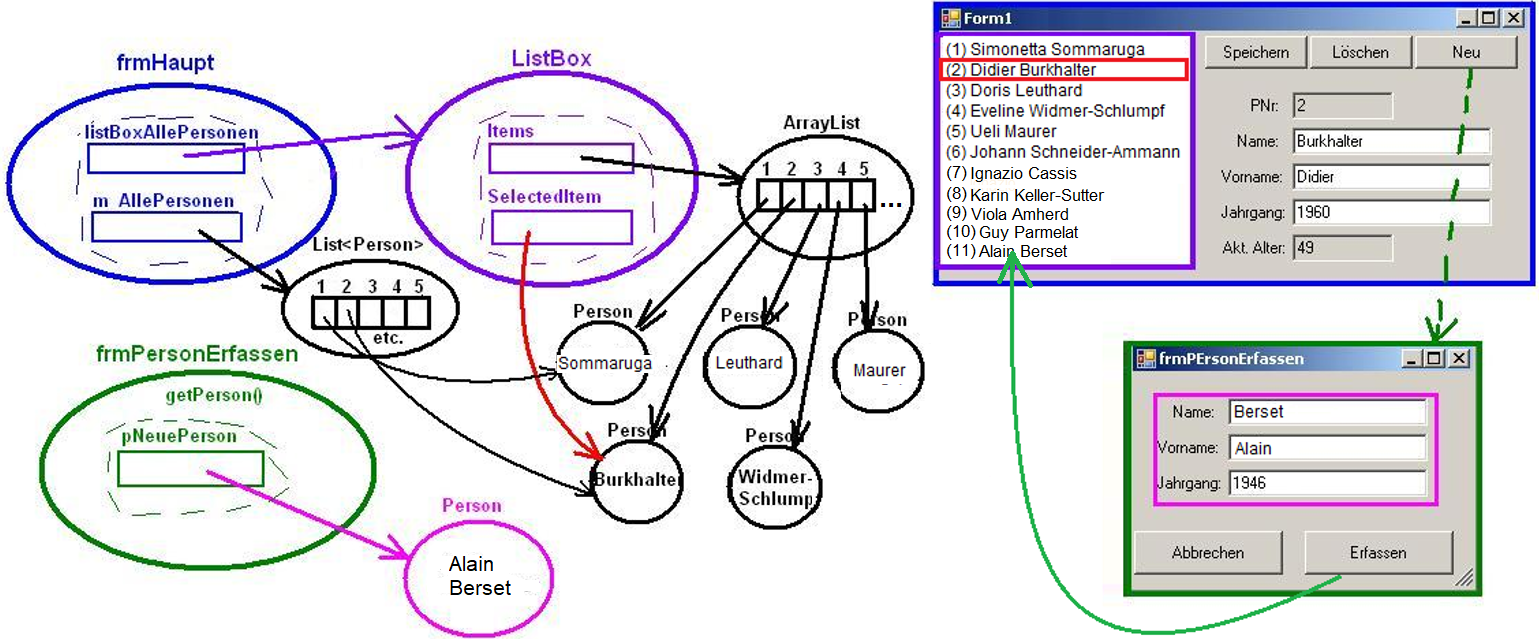
return;

**???**

}

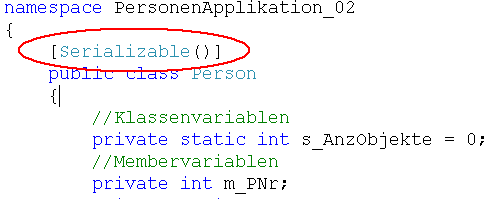
**Schritt 9** (Situation im Arbeitsspeicher aufzeichnen)

Betrachten Sie die Situation im Arbeitsspeicher.



**Schritt 10** (Das aktuelle Array in einer Datei binär, persistent speichern)

Unser Formular enthält ja bekanntlich eine dynamische Liste mit der Bezeichnung m\_AllePersonen , welche diverse Personen-Objekte enthält. Wir möchten nun auf das Ereignis “FormClosed“ (Das Formular wurde geschlossen) so reagieren, dass der Inhalt des Arrays in eine Datei gespeichert wird (= persistente Speicherung). Ergänzen Sie dazu über der Klassenbezeichnung der Klasse Person die Information [Serializable()] (siehe unten)



Versuchen Sie danach, analog dem Projekt des letzten Kapitels, das Speichern und Laden der Informationen in, resp. aus einer Datei zu realisieren.

**Schritt 11 (optionale Erweiterung)**

Versuchen sie die Daten im JSON Format verschlüsselt zu speichern. Suchen sie dazu passende Nuget-Packete die sie unterstützen