Zusammenfassung Modul 226

Objektorientiert implementieren

Version: 01.00
Freigabe: 20.05.11

Janik von Rotz
Hoheneich 4, 6064 Kerns
Internet www.janikvonrotz.ch
Email contact@janikvonrotz.ch

Inhaltsverzeichnis

Änderungskontrolle

Version	Datum	Autor	Beschreibung der Änderung	Status
<<#>>	<<Datum>>	<<Name>>		

Referenzierte Dokumente

Nr.	Dok-ID	Titel des Dokumentes / Bemerkungen
<<#>>	<<#>>	<<Titel/Name des Dokumentes>>

Titel:	Zusammenfassung Modul 226	**Typ:**	Hanbuch	**Version:**	01.00
Thema:	Objektorientiert implementieren	**Klasse:**	öffentlich	**Freigabe:**	20.05.11
Autor:	Janik von Rotz	**Status:**	Freigegeben	**PrtDat./gültig bis:**	20.05.11 / Mai 11
Ablage/Name:	c:\Dokumente und Einstellungen\ILZ32\Eigene Dateien\Dropbox\exchange\teil_abschluss_prüfungen\zusammenfassung\m226\modul226_zusammenfassung.docx			**Registratur:**	.

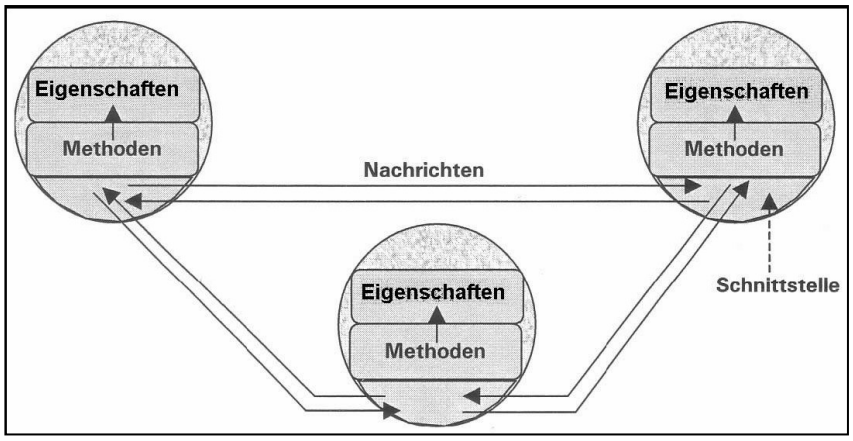
1. Objektorientiert Programmieren

1.1 Probleme strukturiert Programmieren

Verwendung der Strukturvariablen ohne Initialisierung
(z.B. ohne auto1.Richtung =1)
Die Strukturvariable direkt auf ungültige Werte setzen
(z.B. auto1.Richtung=6, MaxGeschw=160 ; Akt-Geschw=420)

- Ist Kapselung durch Funktion möglich >> Man kann immer noch ungehindert direkt auf die Strukturvariablen (z.B: AktGeschw) zugreifen oder die Strukturvariable ohne Initialisierung verwenden.

1.2 OOP Beschreibung

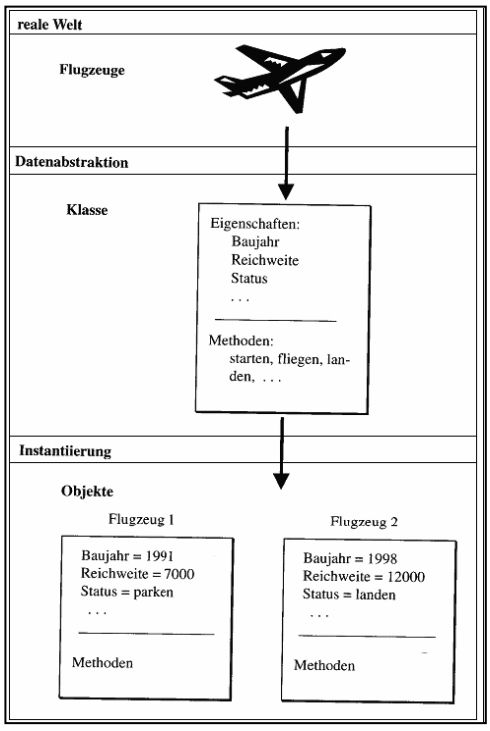


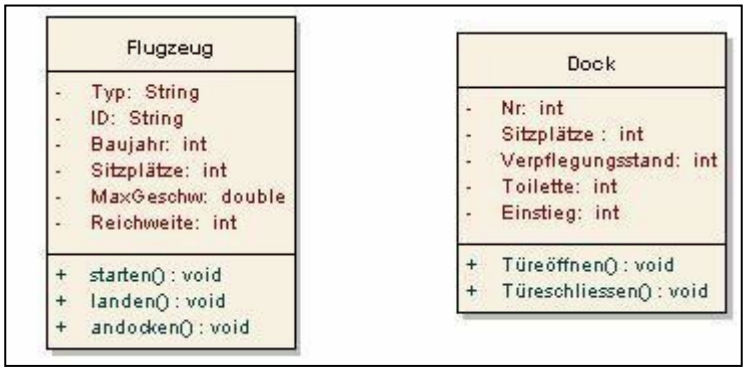
Bei der OOP bilden die Objekte eine Einheit aus:
- Eigenschaften (= Daten)
- Methoden (=Funktionen).

Ein … Objekt …. besteht aus …Eigenschaften / Membervariablen (Daten)… und
…Methoden / Memberfunktion (Funktionen)….

1.3 Klassen und Objekte

Hier eine mögliche Darstellung Abstraktion der Objekte als Theorieform:
UML Notation:





Regeln der OOP

- Aus einer Klasse entstehen Objekte. Dieser Vorgang nennt man Instanzierung.
- Ein Objekt ist eine Instanz einer Klasse
- Aus einer Klasse können beliebig viele Objekte erstellt werden.
- Gehört eine Variable zu einer Klasse, nennen wir Sie Eigenschaft / Instanzvariablen
- Gehört eine Funktion zu einer Klasse, nennen wir Sie Methode
- Die Methoden einer Klasse können auf alle Eigenschaften der Klasse zugreifen, egal ob die Eigenschaften
- public oder private sind.
- Auf public Eigenschaften und Methoden kann von ausserhalb der Klasse zugegriffen werden
- Auf private Eigenschaften und Methoden kann von ausserhalb der Klasse NICHT zugegriffen werden
- Die Aufgabe des Softwareentwicklers ist, Klassen zu erstellen und zu beschreiben bei welchen Aktionen
- Objekte dieser Klassen erstellt werden sollen.
- Während der Programmierung existieren keine Objekte, nur Klassen.
- Während der Programmausführung entstehen Instanzen (Objekte) aus den Klassen.
- Statische Eigenschaften werden beim Programmstart erstellt. Sie exisitieren genau einmal, ganz egal ob
- kein, ein oder hunderte von Objekten der Klasse erzeugt wurden.
- Statische Methoden dürfen nur auf statische Eigenschaften zugreifen.
- Auf statische Elemente kann man ohne Instanz zugreifen.

1.3.1 Programm.cs

```
namespace GrundBeispielOOP
{
   class Program
   {
      static void Main(string[] args)
      {
         Kreis k1 = new Kreis();
         Kreis k2 = null;
         k2 = new Kreis();

         k1.setDurchmesser(15);
         k2.setDurchmesser(2000);
         double x = k1.getDurchmesser();

         Kreis k3 = new Kreis(56);

         Kreis.MachWas(5);



         //k1.Durchmesser = 15;
         //double x = k1.Durchmesser;
      }
   }
}
```

1.3.2 Kreis.cs

```
namespace GrundBeispielOOP
{
   public class Kreis
   {
      //Klassenvariablen
      private static double Pi = 3.141592654;
      //Membervariablen
      private double m_Durchmesser;

      //Konstruktoren
      public Kreis() {
         setDurchmesser(1);
      }
      public Kreis(double value)  {
         setDurchmesser(value);
      }

      //Membervariablen
      public double getDurchmesser() {
         return m_Durchmesser;

      }
      public void setDurchmesser(double value)  {
         if (value < 1)
            value = 1;
         else if (value > 100)
            value = 100;
         m_Durchmesser = value;
      }
      /*
      public double Durchmesser  {
         get { return m_Durchmesser; }
         set { m_Durchmesser = value; }
      }
      */



      public static void MachWas(double value)
      {
         Pi = value;

      }
   }
}
```

2. Datenbankzugriff

Connection: repräsentiert die Verbindung zur Datenquelle.
Command: repräsentiert eine Abfrage oder einen Befehl, der von der Datenquelle ausgeführt werden soll.
DataReader Mit Hilfe dieser Klasse kann man Daten lesen, diese aber nicht ändern (Readonly).

2.1 Connection Einrichten

private OleDbConnection cnn = new OleDbConnection();

Access (<2007) via OLEDB:
cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\\myDb.mdb;User Id=admin;Password="

Access (>=2007) via OLEDB:
cnn.Open "Provider=Microsoft.ACE.OLEDB.12.0;Data Source=c:\\myDb.accdb; Persist Security Info=False; "

Access via OLEDB ODBC:
cnn.Open "Provider=MSDASQL;Driver={Microsoft Access Driver (*.mdb)};Dbq=c:\ \mydb.mdb;Uid=Admin;Pwd=
"

SQL-Server via OLEDB ODBC:
cnn.Open "Provider=MSDASQL;Driver={SQL Server};Server=myServerName;Database=myDatabaseName;Uid=sa;Pwd=sa"

MySQL via OLEDB:
cnn.Open "Provider=MySQLProv;Data Source=mySQLDB;User Id=root;Password= "

2.2 Befehle ohne Datenrückgabe

Wenn Sie einen Befehl an eine Datenbank senden wollen, müssen Sie zuerst die entsprechende Klasse (SqlCommand oder OleDbCommand, etc.…) instanziieren.
OleDbCommand cmd = new OleDbCommand();

Dem Objekt cmd kann man mittels Eigenschaft CommandText den auszuführenden Befehl mitteilen:
cmd.CommandText = "delete From Personen where Vorname = 'Hans' ";

Dem Objekt cmd, das einen an das DBMS zu sendenden Befehl repräsentiert, müssen wir noch mitteilen, auf welche Datenbank sich dieser Befehl bezieht (=Verbindung)
cmd.Connection = cnn_Sql;

Für Befehle, die uns keine Antworten zurückgeben (also kein Select-Statement) können wir den Befehl mittels der Methode ExecuteNonQuery() an die Datenbank absenden.
cmd.ExecuteNonQuery();

Die Methode ExecuteNonQuery() gibt uns als Rückgabewert die Anzahl betroffener Datensätze zurück .
int k = cmd.ExecuteNonQuery(); MessageBox.show("Von dieser Aktion waren " + k + " Datensätze betroffen.");

2.3 Befehle die Daten zrückgeben (Command & DataReader)

Wenn Sie Daten aus einer Datenquelle lesen möchten, müssen Sie zuerst ein SQL-Befehl an die Datenbank senden. Wir können dazu die zuvor betrachtete Klasse Command verwenden.

```
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "Select * From Personen";
cmd.Connection = cnn_Sql;
```

Bei der Klasse DataReader gilt dasselbe wie bereits bei den Klassen Connection und Command angesprochen wurde Die Klasse ist abstrakt. Wir müssen also die Klasse entsprechend unserer Verbindungsinformationen aussuchen (bei OLEDB OleDbDataReader, etc.)

```
SqlDataReader myReader = cmd.ExecuteReader();
int iPersNr;
String sNameVorname;
while(myReader.Read()==true) {
 iPersNr = myReader.GetValue(0);
sNameVorname = myReader.GetString(1) + " " + myReader.GetString(2)
```

Oder …. über den Spaltennamen zugreifen

```
while(myReader.Read()) {
String csName = myReader["Name"].ToString()
String csVorname = myReader["Vorname"].ToString()
}
```

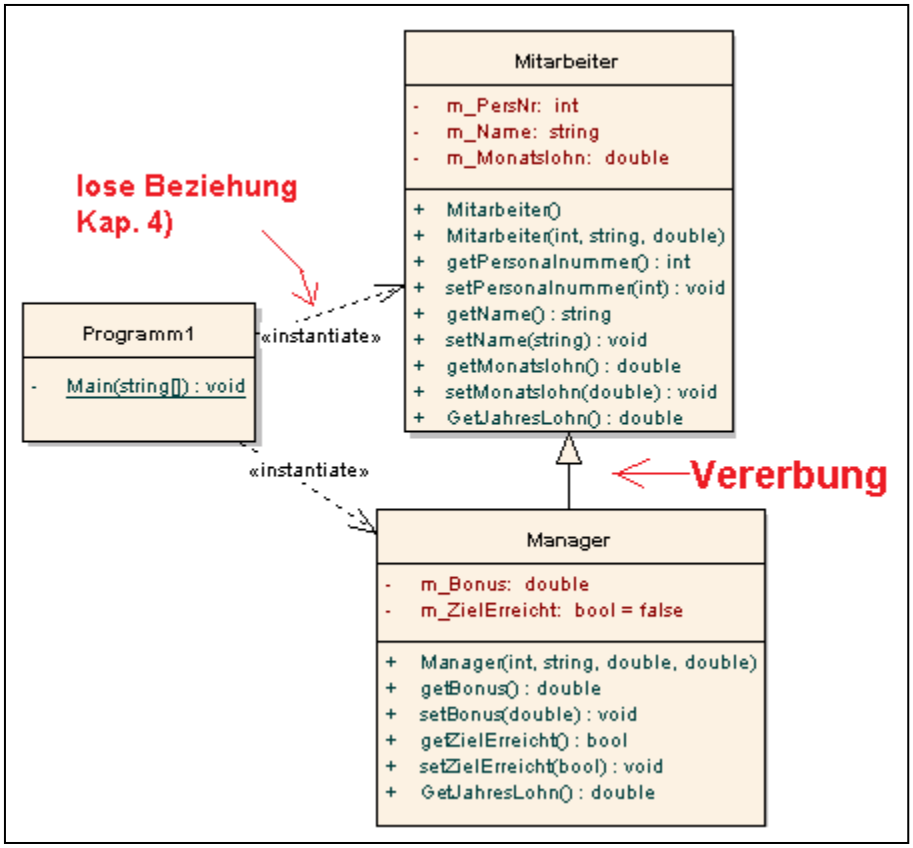
Am Schluss müssen wir den DataReader schliessen

```
myReader.Close();
```

3. Klassenbeziehungen und UML-Klassendiagramm

3.1 Eine Klasse erbt von einer anderen Klasse

UML Diagramm:



Programmcode:

```
public class Programm1
{
    static void Main( string[] args ) {
        Mitarbeiter mi = new Mitarbeiter(222,"Arbeiter Hans", 3800.00);
        Manager ma = new Manager(333, "Manager Peter", 15000, 20000.00);
        ma.setZielErreicht(true);
        System.Console.WriteLine(mi.getName() + ": " + mi.GetJahresLohn().ToString());
        System.Console.WriteLine(ma.getName() + ": " + ma.GetJahresLohn().ToString());
        System.Console.ReadLine(); //Damit das Programm nicht autom. endet
    }
}
```

```
public class Mitarbeiter
{
    //Membervariablen
    private int m_PersNr;
    private string m_Name;
    private double m_Monatslohn;
    //Konstruktoren
    public Mitarbeiter() {
        m_PersNr = -1;
        setName("");
        setMonatslohn(0);
    }
    public Mitarbeiter(int PersNr, string Name, double Monatslohn) {
        m_PersNr = PersNr;
        setName(Name);
        setMonatslohn(Monatslohn);
    }
    ... den rest können Sie ja selber implementieren (gem. UML)
}
```

da hier nichts steht, erbt diese Klasse autom. von Object

```
public class Manager : Mitarbeiter
{
    //Eigenschaften
    private double m_Bonus;
    private bool m_ZielErreicht = false;
    //Konstruktoren
    public Manager(int PersNr, string Name, double Monatslohn, double Bonus) {
        setPersonalnummer(PersNr);
        setName(Name);
        setMonatslohn(Monatslohn);
        setBonus(Bonus);
    }
    //Propreties
    public double getBonus() {
         return m_Bonus;
    }
    public void setBonus(double value) {
        m_Bonus = value;
    }
    public bool getZielErreicht() {
        return m_ZielErreicht;
    }
    public void setZielErreicht(bool value) {
        m_ZielErreicht = value;
    }
    new public double GetJahresLohn() {
        if (m_ZielErreicht == true)
            return base.GetJahresLohn() + m_Bonus;
        else
            return base.GetJahresLohn();
    }
}
```

hier findet die Vererbung statt

die Methode wird mit new überschrieben

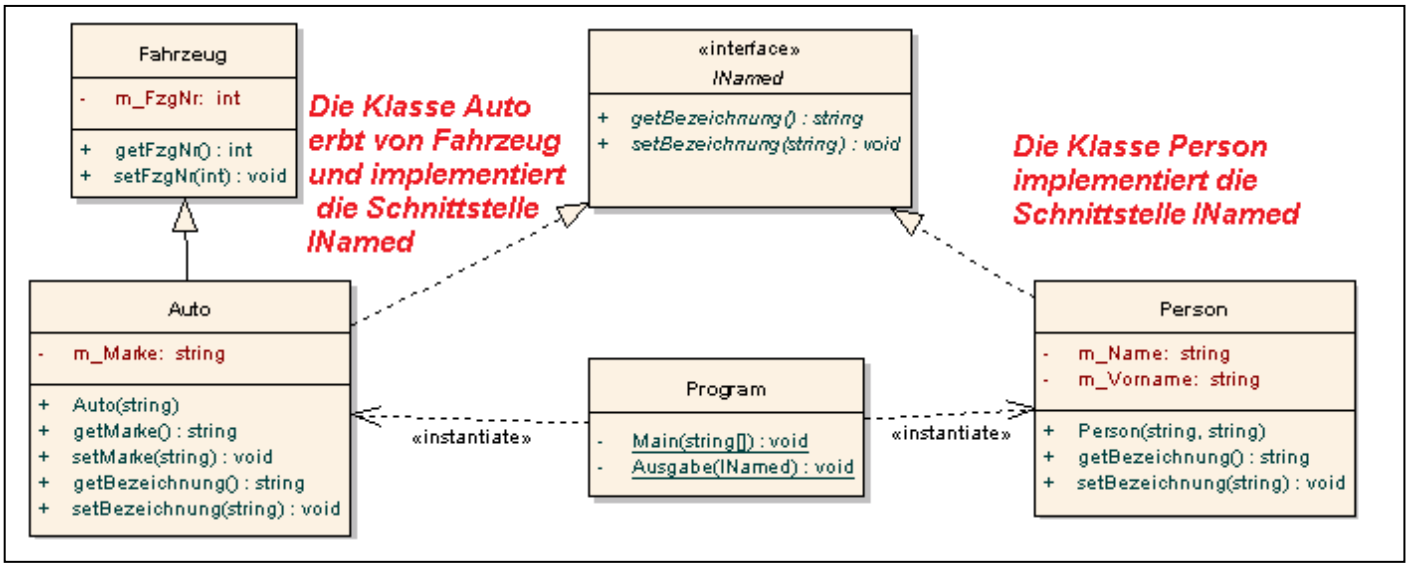
3.2 Eine Klasse implementiert eine Schnittstelle

Schnittstellen (engl. Interfaces) definieren, ähnlich wie Basisklassen, eine Grundfunktionalität. In der Regel handelt es sich dabei um eine Gruppe von sachlich zusammengehörenden Methoden.

Schnittstellen dürfen im Gegensatz zu Basisklassen der Vererbung, **keine Membervariablen** enthalten.
Schnittstellen dürfen auch **keine ausprogrammierten Methoden** enthalten sondern nur eine Art Prototypen. Man nennt diese Prototypen abstrakte Methoden, da Sie noch nicht fertiggestellt wurden.

Eine abstrakte Methode sagt lediglich, wie eine Methode heissen soll, was Sie zurückgibt (Funktionsrückgabewert) und welche Parameter sie benötigt.

3.2.1 UML Diagramm:




```
interface INamed {
    string getBezeichnung();
    void setBezeichnung(string s);
}
```

```
public class Fahrzeug {
    Membervariablen
    private int m_FzgNr;
    //Methoden
    public int getFzgNr() {
         return m_FzgNr;
    }
    public void setFzgNr(int nr) {
        m_FzgNr =nr;
    }
}
```

3.2.2 Klassen

```
public class Person :INamed{
    //Membervariablen
    private string m_Name;
    private string m_Vorname;
    //Konstruktor
    public Person(string sName, string sVorname){
        m_Name = sName;
        m_Vorname = sVorname;
    }
    //Methoden
    public string  getBezeichnung() {
        return m_Vorname + " " + m_Name;
    }
    public void  setBezeichnung(string s) {
        m_Name = s;
    }
}
```

genau wie bei der Vererbung! Das bedeutet, Person implementiert die Schnittstelle INamed

Diese Methoden müssen folglich "ausprogrammiert" werden

```
public class Auto: Fahrzeug, INamed {
    //Membervariablen
    private string m_Marke;
    //Konstruktor
    public Auto(string Marke) {
        setMarke(Marke);
    }
    //Methoden
    public string getMarke() {
        return m_Marke;
    }
    public void setMarke(string Marke){
        m_Marke = Marke;
    }
    public string getBezeichnung(){
        return getMarke();
    }
    public void setBezeichnung(string s){
        setMarke(s);
    }
}
```

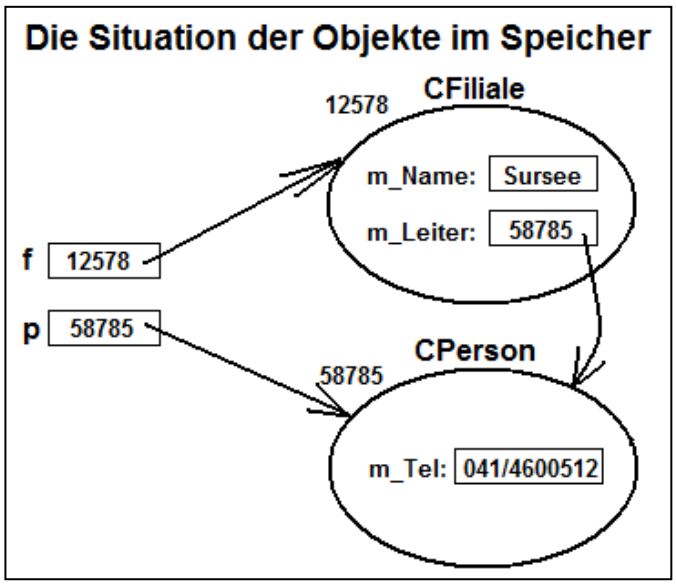
Die Klasse Auto erbt von Fahrzeug und implementiert die Funktionalität der Schnittstelle INamed

Diese Methoden müssen folglich "ausprogrammiert" werden

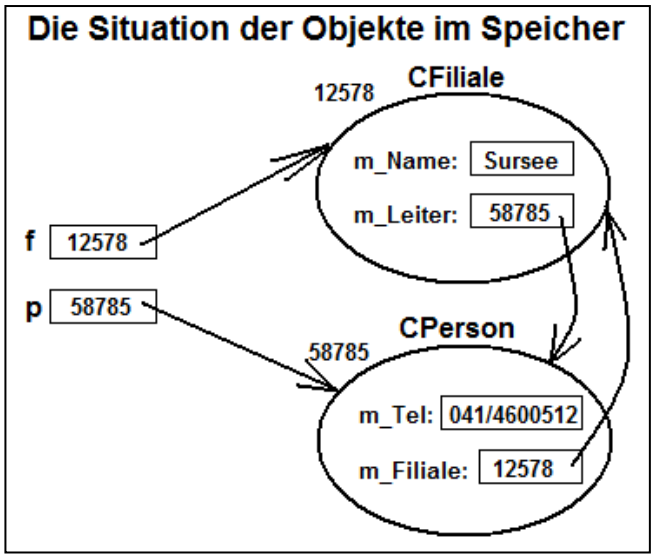
```
class Program {
    static void Main(string[] args) {
        Person p = new Person("LaForge", "Geordi");
        INamed n = new Person("Kirk", "James");

        Auto a = new Auto("Volkswagen");
        a.setFzgNr(12);
        INamed n2 = a;

        Ausgabe(p);
        Ausgabe(n);
        Ausgabe(a);
        Ausgabe(n2);

        Console.ReadLine();
    }

    static void Ausgabe (INamed n) {
        //Diese Methode akzeptiert als Parameter irgendein Objekt, welches die
        //Schnittstelle INamed implementiert.
        Console.WriteLine(n.getBezeichnung());
    }
}
```

3.3 Ein Objekt speichert die Adresse eines anderes Objektes

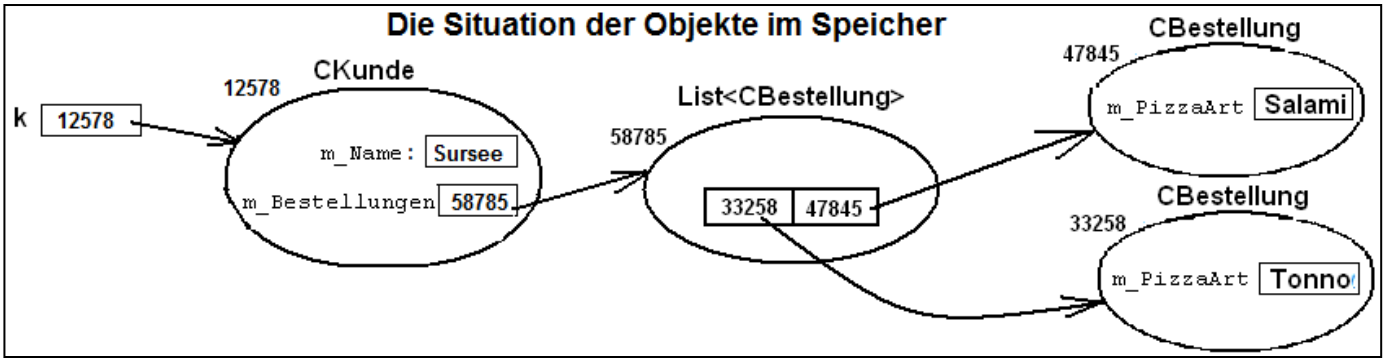
3.3.1 Unidirektionale Assoziationen



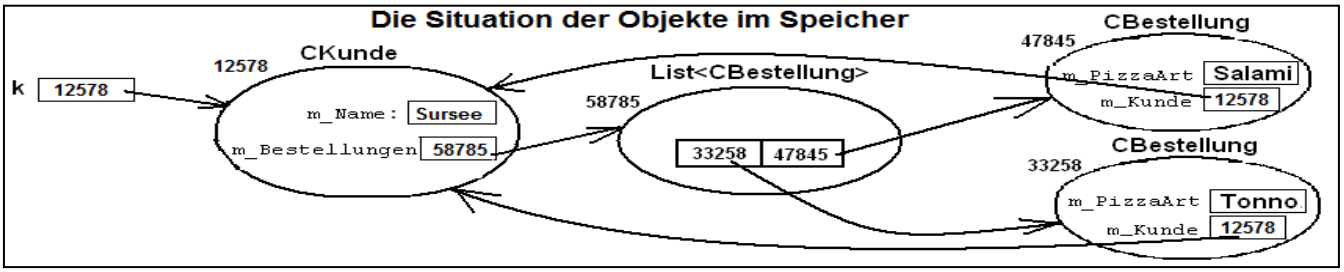
3.3.2 Bidirektionale Assoziationen



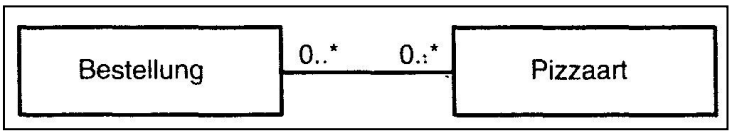
3.3.3 Gerichtete 1:n Beziehungen



3.3.4 Bidirektionale 1:n Beziehungen

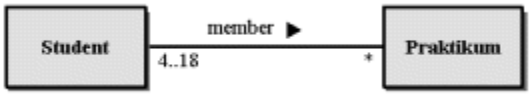


3.3.4.1 Beziehungen des Typs n:m



3.4 Beziehungstypen

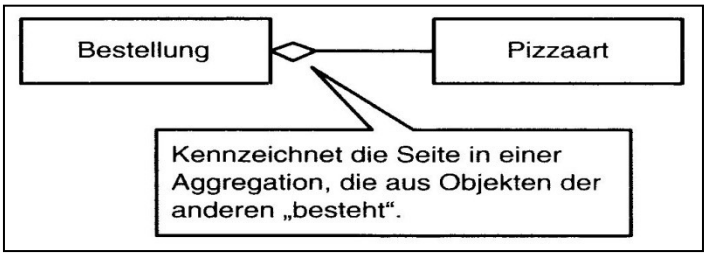
3.4.1 Assoziation



- Zwischen den beteiligten Klassen bestehenkeine essenziellen Abhängigkeiten, d.h., ihreObjekte 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..*

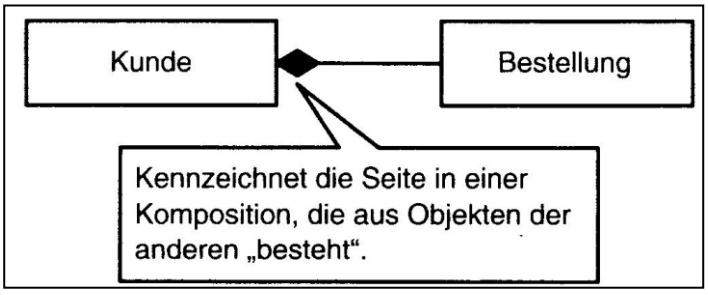
3.4.2 Aggregation

existenzunabhängige “hat“- oder “ist-Teil-von“-Beziehung



3.4.3 Komposition

existenzabhängige “hat“- oder “ist-Teil-von“-Beziehung



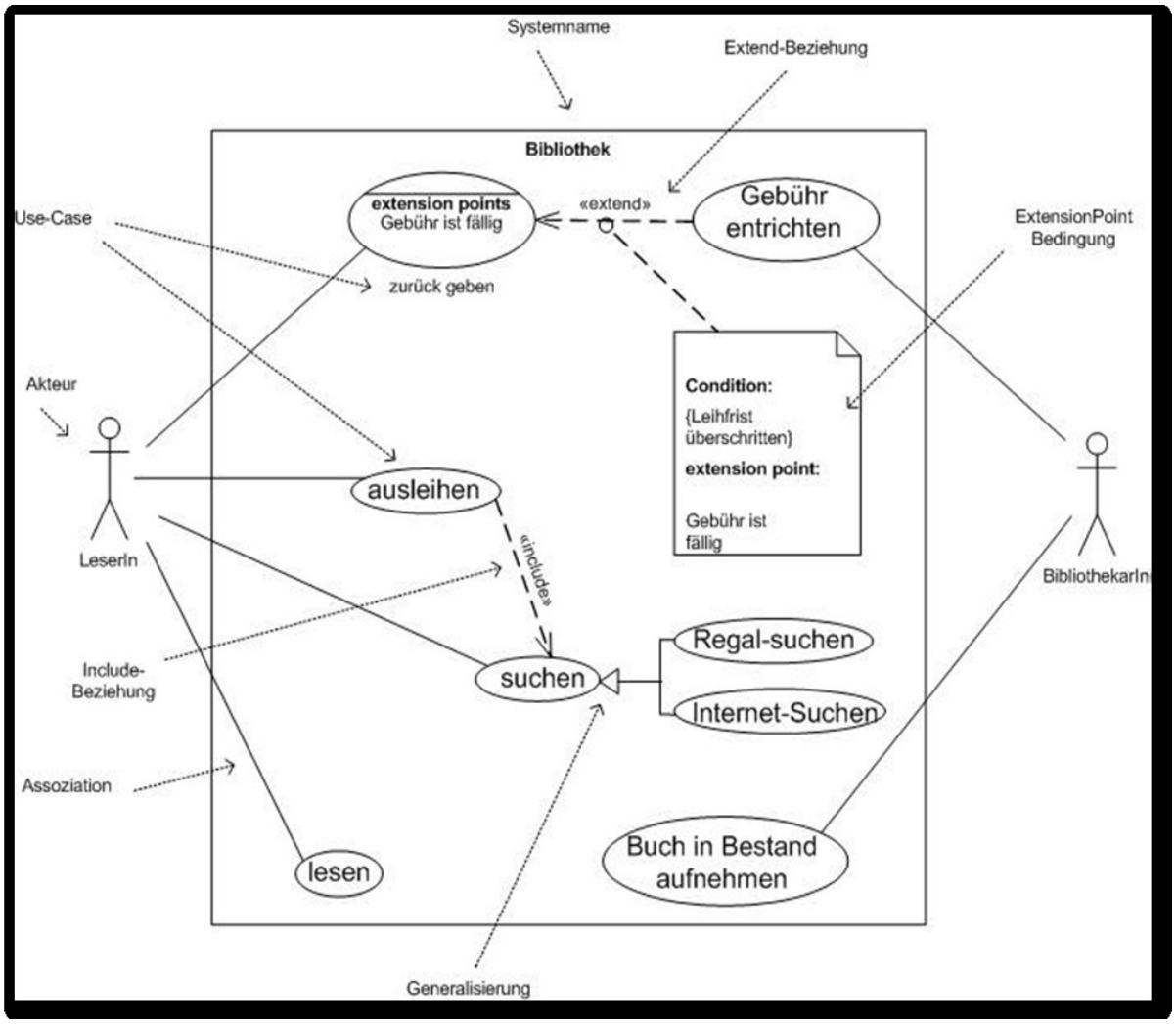
4. UML Diagramme

4.1 UML-Use-Case-Diagramm (Anwendungsfalldiagramm)

4.1.1 Zweck

- Modelliert die Funktionalität des Systems.
- Nur Anwendungsfälle, die für den externen
- Betrachter wahrnehmbar sind und einen
- nutzen erbringen.
- Nicht was im System eigentlich geschieht,
- sondern was der Anwender vom System
- erwarten kann.
- Es gibt keine Reihenfolge
- Anwenderwünsche werden erfasst und
- dokumentiert. Anwender kommentiert das
- Diagramm.
- Sollen möglichst einfach gehalten werden.

4.1.2 Übersicht



4.1.3 Notationselemente

Systemgrenze (System Boundary)
System, dass die benötigten Anwendungsfälle bereitstellt.
Alle Elemente innerhalb des Systems stellen Bestandteile des Systems dar.
Ob man die Systemgrenze zeichnet, ist nicht obligatorisch.

Akteur (Actor)
Typ oder Rolle, die ein externer Benutzer oder ein externes System.
Werden ausserhalb des Systems gezeichnet.
Man darf selber aussagekräftige Symbole verwenden.

Anwendungsfall (Use Case)
Abgeschlossene Menge von Aktionen im System.
Liefern erkennbaren Nutzen
Die Funktionalität wird gezeigt und nicht wie genau etwas funktioniert.

Assoziation (Association)
Beziehung zwischen Akteuren und Anwendungsfällen
Kardinalitäten werden angegeben (1 : 1...*, etc...)
Ungerichtete Assoziation: Die Kommunikationsrichtung ist unspezifiziert. Jeder kann mit jedem reden.
Gerichtete Assoziation: Weg der Kommunikation. Einwegkommunikation.
Alle im Anwendungsfall vorkommende Personen müssen vorhanden sein für die Ausführung.

Generalisierung / Spezialisierung (Generalization)
Kann mit Akteuren und mit Anwendungsfällen geschehen.
Anwendungsfälle mit ähnlichen Funktionen können hierarchisch zugeordnet werden und wiederverwendet werden.

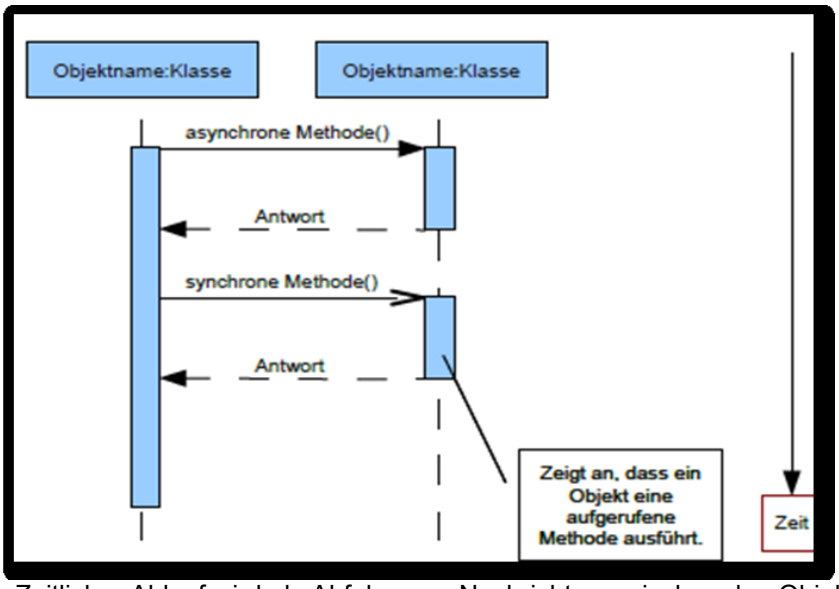
Include-Beziehung (Include Relationship)
Verknüpft einen Anwendungsfall mit einem anderen Anwendungsfall, der zwingend zusätzlich ausgeführt werden muss.

Extend-Beziehung (Exclude Relationship)
Verknüpft einen Anwendungsfall mit einem anderen Anwendungsfall, der nicht zwingend zusätzlich ausgeführt werden muss.
Es müssen Bedingungen eingefügt werden. Diese werden mit condition benannt.
Man muss extensions points setzen (diese kommen zweimal vor!).

Bei Beziehungen sollte man keinen „Teufelskreis“ entstehen lassen!

4.2 UML-Sequenzdiagramme

Mit einem UML-Sequenzdiagramm kann man die Kommunikation von Objekten beschreiben.



Zeitlicher Ablauf wird als Abfolge von Nachrichten zwischen den Objekten dargestellt. Die Zeit schreitet von oben nach unten fort.

4.2.1 Beispiel