Dokumentation BT-S

Teil 2 - Wichtige Begrifflichkeiten der Programmierung in C#

Gemeinsamkeiten von Arrays und Listen

- Sind Objekte (in C#)
- Sind nullbasiert (Die erste Stelle ist Stelle 0, die zweite Stelle 1, usw.)
- Ermöglicht Speicherung von mehreren Variablen des gleichen Typs oder gleichen Objekten
- Deklaration durch Angabe des Typs (`type[] arrayName` und `List<type> listName`)
- Die einzelnen Elemente können direkt bei Erstellung oder im Nachhinein definiert werden:
  - Leeres Array von „int“-Elementen mit 5 Stellen:
    ```
    int[] array = new int[5];
    ```
    Befüllung im Nachhinein mit: `array[0] = 5;`
    (an der ersten Stelle wird ein „int“ mit dem Wert „5“ eingefügt)

  - Volles Array mit 5 Stellen:
    ```
    int[] array2 = { 1, 3, 5, 7, 9 };
    ```

  - Leere Liste von „string“-Elementen:
    ```
    List<string> listOfStrings = new List<string>();
    ```
    Befüllen z.B. mit: `listOfStrings.Add("a string");`
    (Am Ende der Liste wird ein „String“ mit dem Inhalt „a string“ eingefügt)

  - Liste mit 3 Stellen erstellen:
    ```
    List<string> listOfNames = new List<string>()
    {
    "John Doe",
    "Jane Doe",
    "Joe Doe"
    };
    ```

Spezielles über Arrays

- Schreibweise: `type[] arrayName` (z.B. `string[] namenMitarbeiter`)
- Es gibt eindimensionale, mehrdimensionale und verzweigte Arrays:
  - **Eindimensional**: Eine Aufzählung der Variablen (`type[] arrayName`)

  - **Mehrdimensional**:
    - Jedes Element der Aufzählung beinhaltet selbst eine Aufzählung (bei zwei Dimensionen)
    - Bei jeder hinzukommenden Dimension erhält jedes Aufzählungselement des Aufzählungselements eigene Aufzählungselemente und das so oft/lange man möchte
    - In der Regel wird maximal ein dreidimensionales Array verwendet, da es bei mehreren Dimensionen schnell zu Fehlern kommen kann (Je mehr Dimensionen, desto komplizierter)

- Schreibweise: Zweidimensional: `type[,] arrayName`
  Dreidimensional: `type[,,] arrayName`
  (für jede weitere Dimension ein weiteres Komma in die eckige Klammer)

- **Verzweigt**: Ein Array von Arrays (`type[][] arrayName`)

- Die Anzahl der Dimensionen und deren jeweilige Länge werden bei Erstellung der Instanz des Arrays festgelegt und sind danach nicht mehr zu ändern
- Wichtige/nützliche Methoden der Array-Klasse:
  - „`Array.Sort()`“ zum Sortieren nach gewünschtem Kriterium
  - „`Array.Find()`“ zum Durchsuchen nach einem bestimmten Element
  - „`Array.Copy()`“ - Kopiert einen Bereich von Elementen eines Arrays in ein anderes Array
  - „`arrayName[zu ändernde Stelle]`“ um eine bestimmte Stelle im Array zu ändern

Spezielles über Listen

- In der Regel wird die Liste „`List<T>`“ verwendet
- Schreibweise: `List<type> listName` (z.B. `List<string> namenMitarbeiter`)
- Eine Liste hat keine vorher festgelegte Größe, es können immer weitere Elemente hinzugefügt werden
- Wichtige/nützliche Methoden der List-Klasse:
  - „`List.Add(Element)`“ zum Hinzufügen eines Elements am Ende der Liste
  - „`List.Insert(Stelle, Element)`“ um ein Element an einer bestimmten Stelle
    1. einzufügen
  - „`List.Remove(Element)`“ um ein bestimmtes Element zu löschen
  - „`List.RemoveAt(Stelle)`“ um eine bestimmte Stelle zu löschen
  - „`List.Sort()`“ zum Sortieren nach gewünschtem Kriterium

Verzweigungen

- Je nachdem welche Eingaben das Programm erhält reagiert es anders.
  Es wird also im Programm verzweigt, um eine entsprechende Aktion auszuführen.
- Eine Verzweigung innerhalb eines Programms wird durch eine Bedingung entschieden.
  - Z.B. Wenn der Benutzer X eingibt, mache A.

- Verzweigungen sind:
  - **„if“-Anweisung**
    - Nach dem „if“ steht in runden Klammern die Bedingung
    - Ist die Bedingung erfüllt, wird der Code darunter (in „{}“) ausgeführt
    - Ist die Bedingung nicht erfüllt, wird der Code unter dem „if“ nicht genutzt und das Programm fährt mit dem restlichen Code fort
    - Z.B.:
```
if(zahl==5)
{
   printf("fuenf\n");
}
```

- **„if-else“-Anweisung**
  - Der „if-Teil“ ist wie bei der „if“-Anweisung
  - Wenn die Bedingung erfüllt ist, wird das „else“ nicht ausgeführt
  - Wenn die Bedingung der „if“-Anweisung nicht erfüllt ist, kann man mit einer „else“-Anweisung Code definieren, der in diesem Fall ausgeführt werden soll
  - Z.B:
```
if(zahl==5)
    {
       printf("fuenf\n");
    }
    else
    {
       printf("nicht fuenf\n");
    }
```
  - In ein „else“ kann man auch immer noch eine weitere „if-else“-Anweisung schreiben. Dies wird aber schnell unübersichtlich, darum ist es sinnvoller bei mehreren möglichen Bedingungen die „if-else if-else“-Anweisung zu verwenden.

- **„if-else if-else“-Anweisung**
  - Statt in jedes „else“ eines „if-else“ ein weiteres „if-else“ zu packen, kann man nach dem „if“ ein „else if“ verwenden, dem man in runden Klammern eine weitere Bedingung angibt.
  - Hierbei wird von oben nach unten jede Bedingung geprüft (von „if“ und „else if“). Wenn eine Bedingung erfüllt ist, wird der entsprechende Code (in „{}“) ausgeführt und die Prüfung von Bedingungen an dieser Stelle beendet. Die „else“-Anweisung wird in diesem Fall nicht ausgeführt.
  - Z.B.:
```
if (zahl==5)
 {
   printf("fuenf\n");
 }
 else if (zahl==3)
 {
   printf("nicht fuenf\n");
 }
 else
 {
   printf("nicht möglich\n");
 }
```

- **„switch-case“-Anweisung**
  - Hier wird hinter „switch“ in runden Klammern der zu überprüfende Wert und in den „case“-Anweisungen der Code festgelegt, der ausgeführt wird, wenn dieser Fall(=case) eintritt, sprich der Wert mit dem Fall übereinstimmt
  - Der Wert kann „hart-codiert“ oder von einer anderen Methode festgelegt werden
  - Nach jedem „case“ folgt ein „break;“, der dafür sorgt, dass der restliche Code nicht ausgeführt wird. Zudem ist ein „default“-Fall sinnvoll, der genutzt wird, wenn keiner der Fälle mit dem Wert übereinstimmt. Ohne „default“-Fall stürzt das Programm an der Stelle ab, wenn keiner der Fälle mit dem Wert übereinstimmt.
  - Z.B.:
```
switch([int] a)
    {
       case 1: printf("a ist eins\n");
       break;
```

```
        case 2: printf("a ist zwei\n");
        break;
        case 3: printf("a ist drei\n");
        break;
        default: printf("a ist irgendwas\n");
        break;
    }
```

- Bei „if“-Anweisungen, „if-else“-Anweisungen und „if-else if-else“-Anweisungen ist die Bedingung nur erfüllt, wenn sie „wahr“ ergibt.
  Dies kann durch sogenannte Vergleichsoperatoren (z.B. „<“, „>“, „==“, „!=“), „hart-codierte“ Werte und/oder Werten bzw. Rückgabewerten von Methoden und Variablen bestimmt werden.
  Es können auch mehrere Bedingungen je „if“ oder „else-if“ festgelegt werden; diese werden mit „&&“ (und) oder „||“ (oder) getrennt und dementsprechend auch gewertet.
  - Z.B.: `if(5 <= [int] zahl)` **oder**
    `if([string] person == „Markus“)` **oder**
    `if(5 < zahl || zahl = 8)`

Schleifen

- Bieten die Möglichkeit einen beliebigen Teil des Codes beliebig oft zu wiederholen
- Es gibt 4 verschiedene Arten, die „while“-, „do-while“-, „for“- und „foreach“-Schleife:
  - **„while“-Schleife**
    - Wird ausgeführt solange die Bedingung hinter „while“ (in runden Klammern) „wahr“ ergibt/erfüllt ist
    - Z.B.:
```
while(number < 5)
    {
        Console.WriteLine(number);
        number = number ++;
    }
```

  - **„do-while“-Schleife**
    - Hier folgt die Bedingung erst am Ende, hinter „while“ in runden Klammern; dadurch wird der Code auf jeden Fall einmal ausgeführt und hinterher überprüft, ob der Code wiederholt werden soll (wenn Bedingung erfüllt ist)
    - Z.B.:
```
do
    {
        Console.WriteLine(number);
        number = number + 1;
    }while(number < 5);
```

  - **„for“-Schleife**
    - Ist kompakter, eine „Alles-in-Einem“-Schleife, da Variable, Bedingung und Werterhöhung (Bestimmung der Wiederholungsanzahl) direkt hinter dem „for“ in runden Klammern angegeben wird und somit nicht an anderer Stelle definiert werden muss
    - Wird wiederholt solange die Bedingung „wahr“ ergibt/erfüllt ist
    - Z.B.:
```
for(int i = 0; i < number; i++)
    }
        Console.WriteLine(i);
    }
```

- **„foreach“-Schleife**
  - Wird ausgeführt für jedes (=for each) gewünschte Element einer Listenart wie ein Array oder eine List<T>
  - Welches Element die „foreach“-Schleife bearbeiten/auslesen/etc. soll, wird hinter dem „foreach“ in runden Klammern angegeben
  - Z.B. :
```
ArrayList list = new ArrayList();
     list.Add("John Doe");
     list.Add("Jane Doe");
     list.Add("Someone Else");

     foreach(string name in list)
     }
        Console.WriteLine(name);
     }
```

Methoden

- Sind eine Art Unterprogramm in einer Klasse und fassen Code zusammen, der von verschiedenen Stellen im Programm aufgerufen werden kann
- Können einen Wert an den Aufrufer zurückgeben
- Verändern die Eigenschaften eines Objekts
- Eine Methode wird nach folgendem Muster angelegt:
  - Zugriffsebene Rückgabewert Bezeichner([optional] Übergabeparameter)
    {
      Auszuführender Code
    }
  - Z.B.:
```
Private void TestMethode(int zahl, string name)
       }
           Console.WriteLine(name);
       }
```

- Zugriffsebenen sind:
  - Public
    - Unbeschränkter Zugriff von überall
  - Protected
    - Zugriff von der enthaltenden Klasse und von erbenden Klassen aus, aber nicht von einer Instanz der Klasse
  - Internal
    - Zugriff von überall innerhalb des enthaltenden Programms/Projekts
  - Protected internal
    - Zugriff von innerhalb des enthaltenden Programms/Projekts innerhalb der enthaltenden Klasse und von davon erbenden Klassen aus
  - Private
    - Zugriff nur von der enthaltenden Klasse aus
  - Private protected
    - Zugriff von der enthaltenden Klasse und von erbenden Klassen aus, aber nicht von einer Instanz der Klasse

- Der Rückgabewert einer Methode bestimmt, welchen Wert die Methode liefert/zurückgibt
  - Mit Hilfe des Schlüsselwortes „return“ im Code der Methode wird ein bestimmter Datentyp (an den Aufrufer) zurückgegeben, welcher mit dem Datentyp in der Methodensignatur übereinstimmen muss
  - Eine Methode kann immer nur einen Wert zurückgeben
  - Z.B.:
```
Private int TestMethode()
      }
        int i = 5;
       return i;
     }
```

- Regeln für den Bezeichner der Methodensignatur (Methodensignatur = Der komplette „Name“ der Methode, der über dem auszuführenden Code steht):
  - Beginnt in der Regel mit einem Großbuchstaben und darf keine Leerzeichen oder Sonderzeichen (außer „_“) enthalten
  - Wenn sie aus mehreren Wörtern bestehen, werden jeweils die Anfangsbuchstaben des Wortes groß geschrieben (z.B. „BerechneSummeAnwesenderLehrer()“)
  - Der exakt gleiche Bezeichner mit denselben Übergabeparametern kann nur einmal verwendet werden, es ist aber möglich eine Methode mit gleichem Bezeichner aber anderen Parametern zu erstellen. Hier wird bei einem Aufruf dann die Methode aufgerufen, bei dem die Übergabeparameter übereinstimmen.

- Aufgerufen wird eine Methode durch ihren Bezeichner plus die runde Klammer dahinter (ggf. mit Werten für die nötigen Übergabeparametern)
  - Z.B.: `TestMethode()` oder `TestMethode(anzahlKlasse, nameLehrer)`
  - Die Reihenfolge für die Übergabeparameter wird in der Methode festgelegt;
    Bei einer abweichenden Reihenfolge muss man vor seinen Wert schreiben, welchen Parameter dieser betrifft
    - z.B. `TestMethode(name: nameLehrer, zahl: anzahlKlasse)`
  - Ob eine Methode an der gewünschten Stelle aufgerufen werden kann, bestimmt die Zugriffsebene

Lesen von MySql-Datenbanken

- Daten aus MySql-Datenbanken können entweder über sogenannte Datenbankbefehle oder in/aus einer Anwendung heraus gelesen werden
- **Lesen mit Hilfe von Datenbankbefehlen:**
  - Werden mit Hilfe unterschiedlicher Programme ausgeführt, die für die Verwaltung von Datenbanken entwickelt wurden
    - Z.B.: Visual Studio Code, Microsoft SQL Server Management Studio

    (Nachdem man sich mit der Datenbank verbunden hat mittels Name, Port, Adresse des Servers und einem gültigen Nutzernamen und Passwortes)
  - Hierbei kann der Anwender Informationen/Datensätze aus der Datenbank einsehen

  - Die wichtigsten Datenbankbefehle für das Lesen von MySql-Datenbanken sind:
    - **SELECT** columnName (,columnName,…) **FROM** tableName
      (Zeigt die Spalte/n von der gewünschten Tabelle)
    - **WHERE** cloumnName=‘value‘
      (In Verbindung mit **SELECT FROM**, zeigt nur die Werte, die der „where“-Bedingung entsprechen)

- Zusätzliche Operatoren für eine „where“-Bedingung:
  - **<>** oder **!=** für „ungleich“
  - **>** „größer als“, **<** „kleiner als“
  - **>=** „größer oder gleich“ **<=** „kleiner oder gleich“
  - **BETWEEN** „alles in diesem Bereich“
    (z.B. **BETWEEN** value1 **AND** value2)
  - **LIKE** „Suche nach einem Schema“
    (z.B. **LIKE** ‘1%‘ „beginnt mit 1“, **LIKE** ‘%1%‘ „enthält eine 1“ oder **LIKE** ‘1_‘ „beginnt mit 1 und ist zwei Stellen lang“)
  - **IN** „ermöglicht die Angabe von mehreren Bedingungen“
    (z.B. **WHERE** columnName **IN** (‘value‘, ‘value‘,…)

- **Lesen der Datenbank aus einer Anwendung heraus:**
  - Mit Hilfe von bereits installierten oder hinzugefügten Erweiterungen (z.B. NuGetPackages) kann durch die mitgelieferten Methoden eine Verbindung zur Datenbank hergestellt und diese dann ausgelesen werden
  - Für unsere Anwendung hat ein Entity-Framework Verwendung gefunden (siehe Quellen – Informationen über Lesen/Schreiben von/zu Datenbanken)
  - Hierbei liest die Anwendung die Informationen aus der Datenbank, um sie für verschiedene Methoden/Felder/etc. verwenden zu können. Will der Anwender diese direkt einsehen, muss er sich dazu eine separate Methode schreiben.
  - Mit folgenden Methoden liest unsere Anwendung aus der Datenbank:
    - Für die Verbindung zur Datenbank benötigt die Anwendung eine Klasse (meist „Context“ genannt, bei unserer Anwendung „EmaContext“), in der die nachfolgende Punkte definiert werden
    - Zunächst wird der Anwendung mitgeteilt, wie sie sich mit der Datenbank verbinden kann:

```
… OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
      optionsBuilder.UseMySQL("server=localhost;database=Ema-
      Database;user=user;password=EmaPassWord");
}
```

    - Anschließend werden mit den Models der Anwendung (= Gegenstück der Datenbank -> Klassenname = Tabellenname, Properties = Spaltennamen) sogenannte „DbSet“s erstellt, damit die Anwendung weiß, welche Tabelle der Datenbank mit welchem Code verbunden werden soll

      Z.B.: `public DbSet< Dealer> Dealers { get; set; }`

    - Um Verbindungsfehler zu vermeiden, wird nun noch jeweils angegeben, wie die genaue Bezeichnung der Datenbanktabelle und dessen PrimaryKey-Spalte lautet

      Z.B.:

```
modelBuilder.Entity<Dealer>(x =>
{
      x.ToTable("dealer").HasKey(k => k.DealerID);
      x.Property(p => p.DealerID).HasColumnName("ID");
}
```

    - Nun können wir mit Hilfe einer Methode an der gewünschten Stelle im Code Daten aus der Datenbank abrufen

      Z.B.:

```
private void InitItems()
{
      var context = new EmaContext();
      Items = context.Items.Include(x => x.Dealer).ToList();
}
```

Schreiben zu MySql-Datenbanken

- Daten für MySql-Datenbanken können entweder über sogenannte Datenbankbefehle oder aus einer Anwendung heraus in die Datenbank geschrieben werden
- **Schreiben zur Datenbank mit Hilfe von Datenbankbefehlen:**
  - Werden mit Hilfe unterschiedlicher Programme ausgeführt, die für die Verwaltung von Datenbanken … (siehe oben, „Lesen von MySql-Datenbanken“, „Lesen mit Hilfe von Datenbankbefehlen“)
  - Hierbei kann der Anwender Informationen/Datensätze in die Datenbank schreiben
  - Die wichtigsten Datenbankbefehle für das Schreiben zu MySql-Datenbanken sind:
    - **ALTER TABLE** tableName
      (Bearbeitet die gewünschte Tabelle)
    - **ALTER COLUMN** columnName datatype;
      (Ändert den Datentyp der Spalte)
    - **ADD**
      (In Verbindung mit **ALTER TABLE**, fügt eine Spalte einer existierenden Tabelle hinzu, z.B. **ALTER TABLE** tableName **ADD** columnName datatype;)
    - **INSERT INTO** tableName (columnName1, columnName2, …)
      VALUES (‘value1‘, ‘value2‘, …);
      (Setzt bestimmte Werte in jeweilige Spalten ein; value1 in column1, usw.)
    - **UPDATE** tableName **SET** columnName = ‘value‘, columnName = ‘value‘, …
      WHERE columnName = ‘value‘;

- **Schreiben zur Datenbank aus einer Anwendung heraus:**
  - Mit Hilfe von bereits installierten oder hinzugefügten Erweiterungen (z.B. NuGetPackages) kann durch die mitgelieferten Methoden eine Verbindung zur Datenbank hergestellt und können Daten in diese geschrieben werden
  - Für unsere Anwendung hat ein Entity-Framework Verwendung gefunden (siehe Quellen – Informationen über Lesen/Schreiben von/zu Datenbanken)
  - Hierbei schreibt die Anwendung mit Hilfe verschiedener Methoden und Properties/Felder die Informationen in die Datenbank. Will der Anwender diese direkt einsehen, muss er sich dazu eine separate Methode schreiben.
  - Mit folgenden Methoden schreibt unsere Anwendung in die Datenbank:
    - Zunächst muss eine Verbindung zur Datenbank bestehen (siehe oben, „Lesen von MySql-Datenbanken“, „Lesen der Datenbank aus einer Anwendung heraus“)
    - Ändern von bestehenden Daten mit „.Update()“:

```
var contextCustomer = new EmaContext();
contextCustomer.Update(CustomerData.First());

contextCustomer.SaveChanges();
```

    - Hinzufügen eines neuen Eintrags/einer Zeile in eine (bestehende) Tabelle mit „.Add()“:

```
var contextOrders = new EmaContext();
contextOrders.Add(NewOrder);

contextOrders.SaveChanges();
```

- Hinzufügen von mehreren Einträgen/Zeilen (eine Liste von Einträgen/Zeilen) in eine (bestehende) Tabelle mit „.AttachRange()“:

```
var contextOrders = new EmaContext();
contextOrders.AttachRange(OrderedItems);

contextOrders.SaveChanges();
```

- Es können auch mehrere Methoden aufeinander folgen, richtig ausgeführt werden sie alle erst, wenn auch das „.SaveChanges();“ folgt.
  Ohne diesen Befehl werden die Befehle nur temporär ausgeführt, es erfolgt aber kein Abschluss und somit kein dauerhaftes Schreiben in die Datenbank

Teil 3 – ER-Modell und Relationenschema

Am Beispiel unserer Datenbank

Relationenschema allgemein

Ein Relationenschema setzt sich aus dem Namen der Datenbank-Tabelle, den beinhalteten Attributen und dessen Typ zusammen. Dies kann auf unterschiedliche Arten visualisiert werden.

Ein simples Beispiel könnte wie folgt aussehen:

- Allgemeines Beispiel : *{Name der Tabelle} : (Attribut 1: Typ von Attribut 1, Attribut 2: Typ von Attribut 2, …)*
- Beispiel aus unserer Datenbank : *Orders : (ID: int, CustomerDataID: int, OderDate: DateTime, TotalPriceEUR: int)*

Zusätzlich gibt es das Relationenschema auch in Tabellenform:

Name der Tabelle (Allgemeines Beispiel)

Attribut	Attribut - Typ

Relationenschemas unserer Datenbank

Tabelle „Orders“

Orders : (ID: int, CustomerDataID: int, OderDate: DateTime, TotalPriceEUR: int, BillViaAddress: bool, BillViaEMail: bool, Shipping: int)

Orders

ID	int
CustomerDataID	int
OrderDate	DateTime
TotalPriceEUR	int
BillViaAddress	bool
BillViaEMail	bool
Shipping	int

Primary key
Foreign key

Tabelle „OrderedItems“

OrderedItems : (OrdersID: int, ItemsID: int, VolumePack: int, VolumeUnitPack: string, SelledAmount: int, SelledUnit: string, PriceUnitEUR: int, SelledAmountItem: int)

OrderedItems

OrdersID	int
ItemsID	int
VolumePack	int
VolumeUnitPack	string
SelledAmount	string
SelledUnit	string
PriceUnitEUR	int
SelledAmountItem	string

Primary key
Foreign key

Tabelle „Items“

Items : (ID: int, Picture: string, Name: string, Description: string, VolumePack: int, VolumeUnitPack: string, SelledAmount: int, SelledUnit: string, PriceSelledUnitEUR: int, DealerID: int, DealerItemNumber: int, Availability: string, DeliveryTime: string)

Items

ID	int
Picture	string
Name	string
Description	string
VolumePack	int
VolumeUnitPack	string
SelledAmount	int
SelledUnit	string
PriceSelledUnitEUR	int
DealerID	int
DealerItemNumber	int
Availability	string
DeliveryTime	string

Primary key
Foreign key

Tabelle „Dealer“

Dealer : (ID: int, CompanyName: string, ContactPerson: string, Street: string, HouseNumber: string*, ZipCode:* string*, City: string, Country: string, PhoneNumber:* string*, EmailAddress: string, Website: string, MinimumOrderValueEUR: int, FreeDeliveryFromEUR: int, StandardDeliveryDeEUR: int)*

Dealer

ID	int
CompanyName	string
ContactPerson	string
Street	string
HouseNumber	string
ZipCode	string
City	string
Country	string

Primary key
Foreign key

PhoneNumber	string
EmailAddress	string
Website	string
MinimumOrderValueEUR	int
FreeDeliveryFromEUR	int
StandardDeliveryDeEUR	int

Tabelle „CustomerData“

CustomerData : (ID: int, CompanyName: string, ContactPerson: string, Street: string, HouseNumber: string*, ZipCode:* string*, City: string, Country: string, PhoneNumber:* string*, EmailAddress: string)*

CustomerData

ID	int
CompanyName	string
ContactPerson	string
Street	string
HouseNumber	string
ZipCode	string
City	string
Country	string
PhoneNumber	string
EmailAddress	string

Primary key
Foreign key

Tabelle „CustomerLoginData“

CustomerLoginData : (ID: int, CustomerID: int, Username: string, Password: string)

CustomerLoginData

ID	int
CustomerID	int
Username	string
Password	string

Primary key
Foreign key

ER-Modell unserer Datenbank