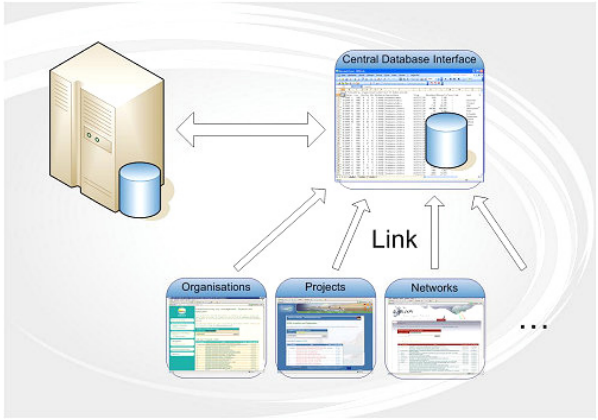
Datenzugriff mit ADO.NET



- Einführung
- Verbindungsorientierter Datenzugriff
- Verbindungsloser Datenzugriff
- Datenbankverbindung mit DataAdapter
- Integration mit XML
- Persistente Objekte

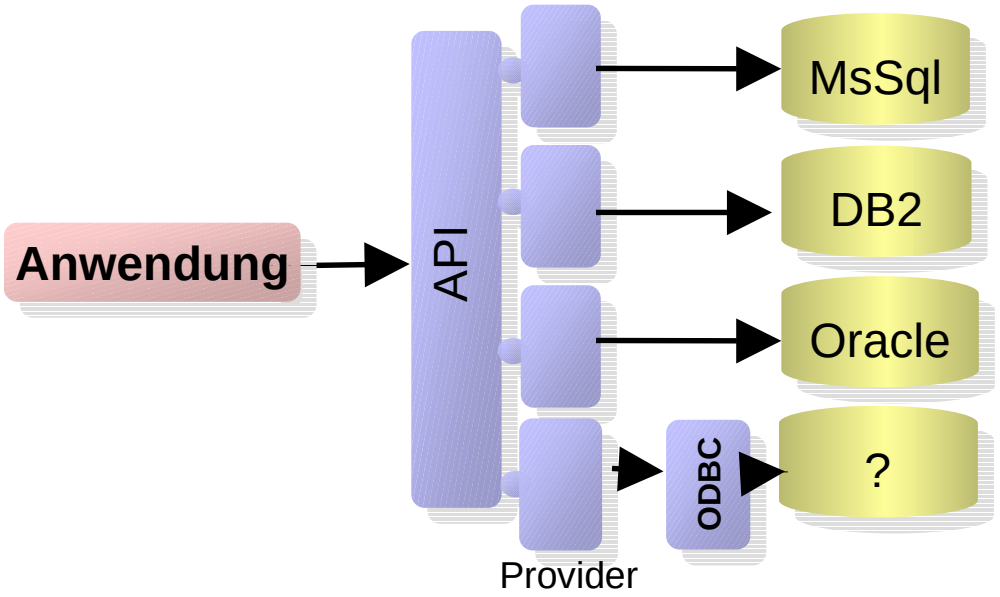
Einführung

ADO.NET

- ADO.NET ist die .NET-Technologie für Zugriff auf strukturierte Daten
- einheitliche objektorientierte Schnittstelle für unterschiedliche Datenquellen
  - relationale Datenbanken
  - XML-Daten
  - andere Datenquellen
- konzipiert für verteilte Anwendungen und Web-Anwendungen
- bietet 2 unterschiedliche Modelle für Datenzugriff
  - verbindungsorientiert
  - verbindungslos

Idee des universellen Datenzugriffs

- Verbindung der (objektorientierten) Programmiersprachen und (relationalen) Datenbanken
- einheitliches Programmiermodell und API
- spezielle Implementierungen für unterschiedliche Datenquellen (*Providers*)



Microsofts Geschichte des Datenzugriffs

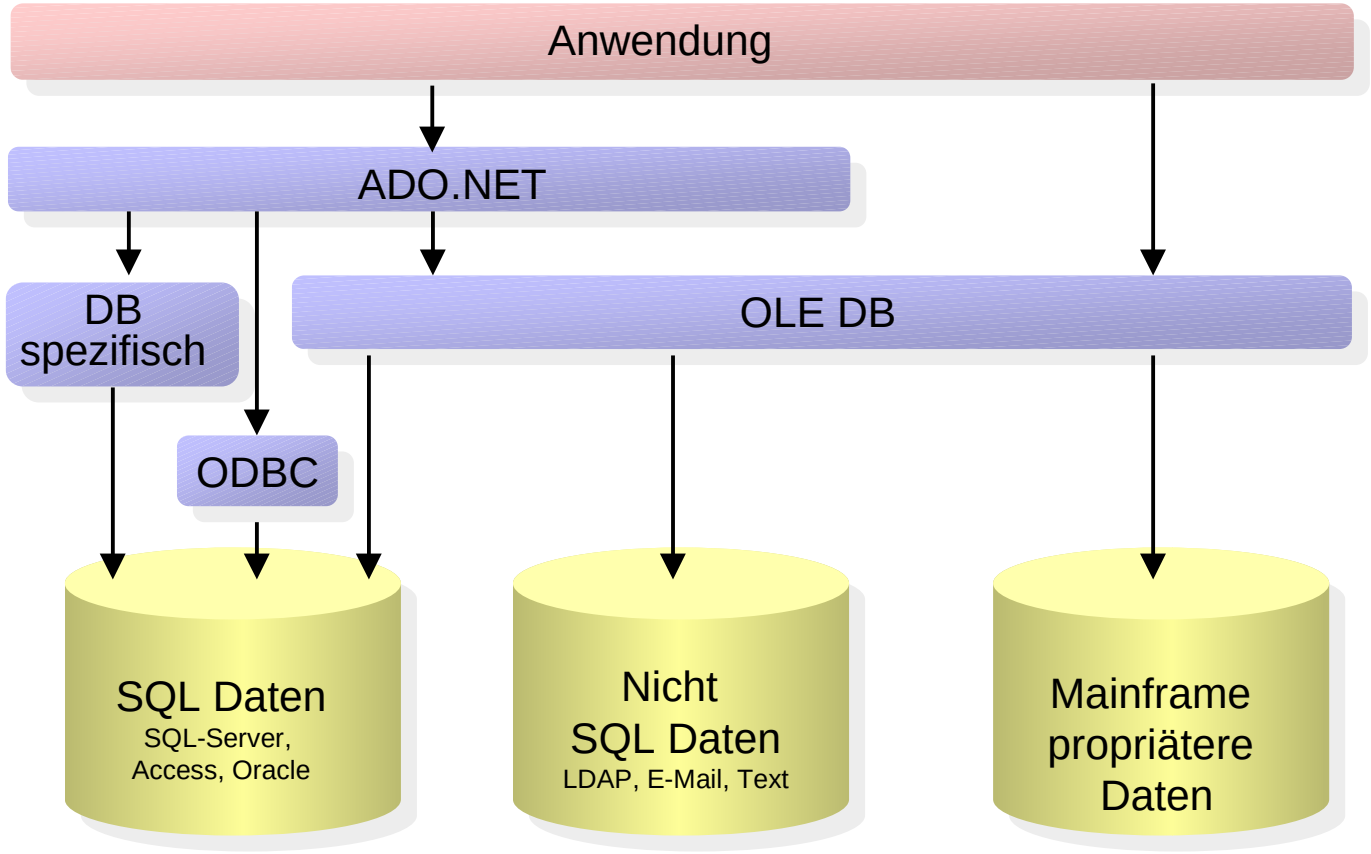
ODBC
-> OLE DB
-> ADO
-> ADO.NET

Web Applikationen benötigen andere Architektur

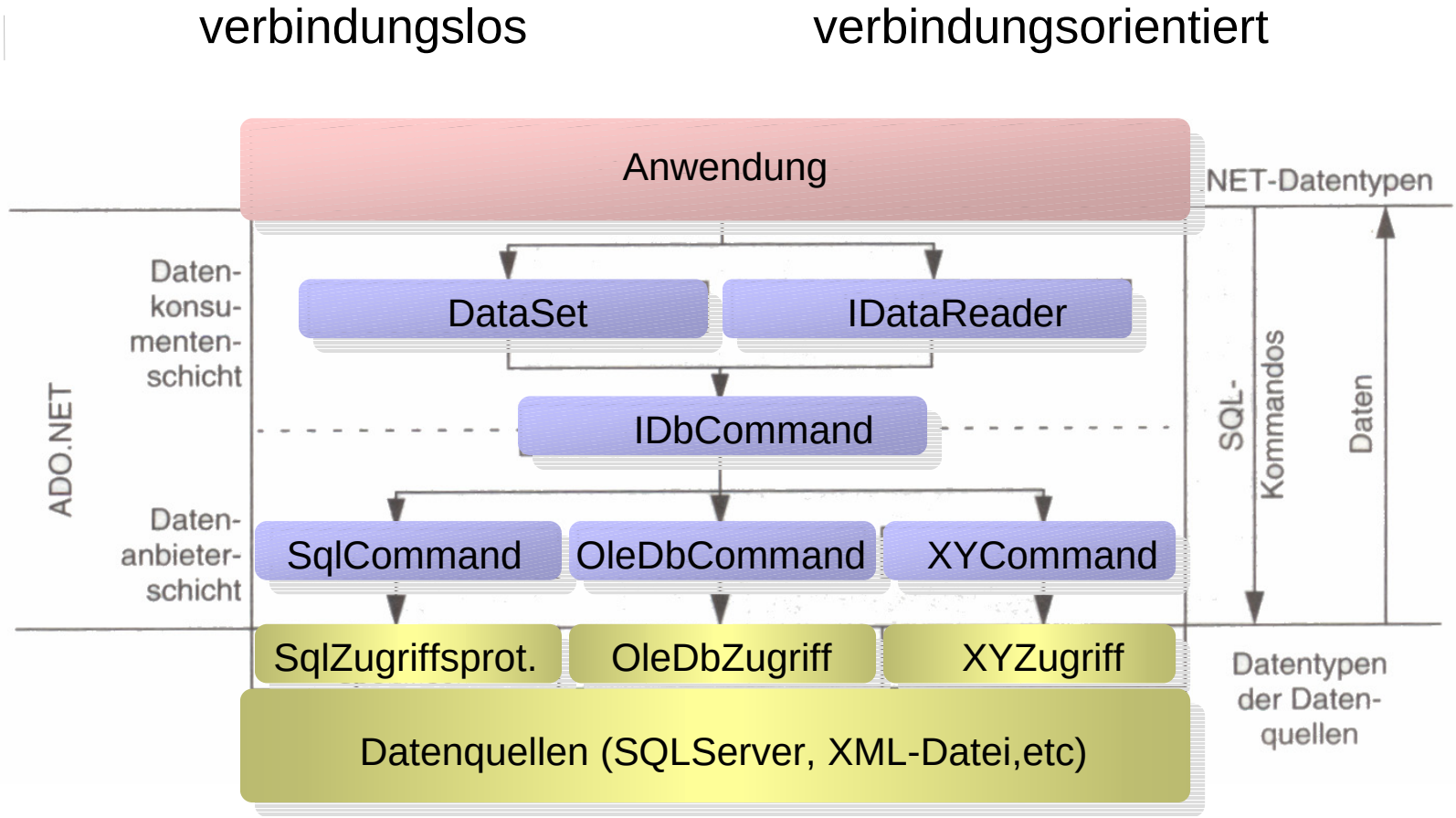
- Lose Kopplung zwischen Applikation und Daten
- XML entwickelt sich zum universellen Datenformat
- Probleme mit bestehenden API's
  - ADO, OLE DB und ODBC: wurden für enge Kopplung und dauerhafte (connected) Verbindungen konzipiert.
  - Es fehlen: Remote Data Services (RDS): für nichtverbundene (disconnected) Verbindungen ohne Zustandsverwaltung zwischen request/response.
  - Es fehlt: Unterstützung für relationales und hierarchisches Datenmodell (XML)

Data Providers

Schichtenmodell der Microsoft Architektur für den Datenzugriff



Architektur von ADO.NET



Verbindungsorientiert versus verbindungslos

School of Engineering

- Verbindungsorientiert
  - Verbindung zur Datenquelle bleibt erhalten
  - Typische Verwendung:
    - *kurze Zugriffe (short running transactions)*
    - *wenige parallele Zugriffe*
    - *immer aktuelle Daten*

- Verbindungslos
  - Keine permanente Verbindung zur Datenquelle
  - Daten im Hauptspeicher zwischengespeichert (Cache)
  - Änderung im Hauptspeicher ≠ Änderung in Datenquelle
  - Typische Verwendung:
    - *viele parallele, lange, lesende Zugriffe (z.B.: Web-Anwendungen)*

ADO.NET Assembly und Namespaces

School of Engineering

Assembly

- System.Data.dll

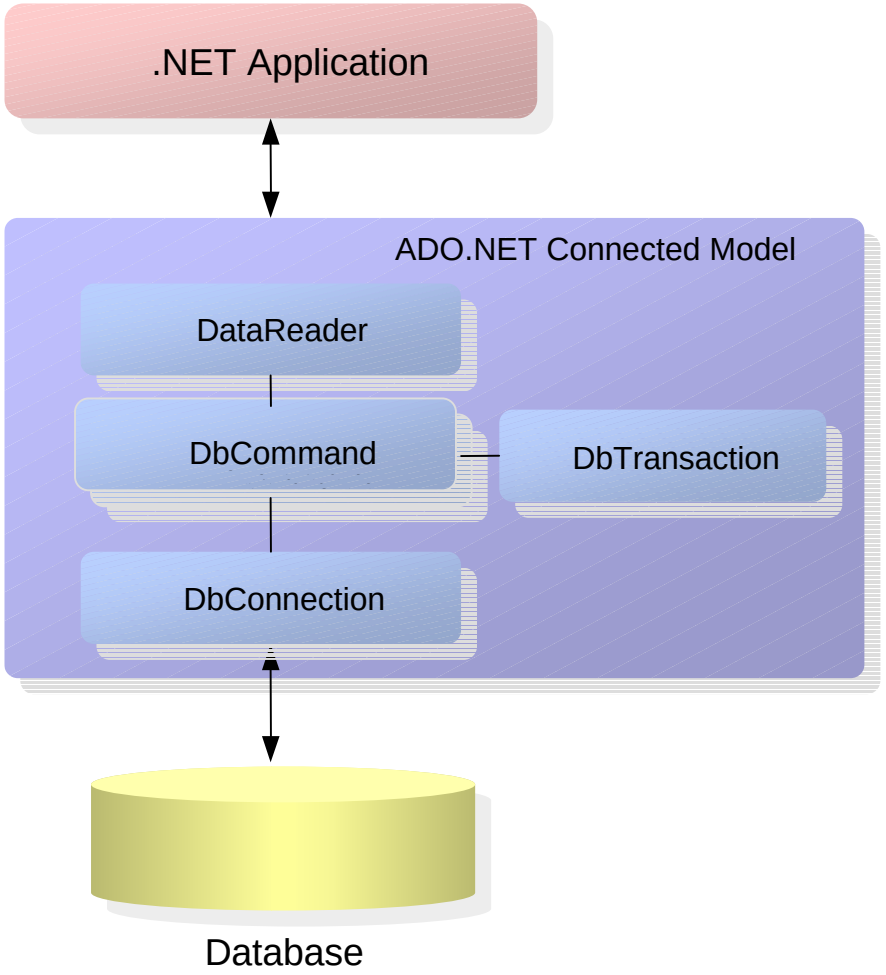
Namespaces:

System.Data	allgemeinen Typen
System.Data.Common	allgemeine Klassen für Treiberimplement.
System.Data.OleDb	OLE DB-Anbieter
System.Data.SqlClient	Microsoft SQL Server-Anbieter
System.Data.Odbc	ODBC-Anbieter (seit .NET 1.1)
System.Data.OracleClient	Oracle-Anbieter (seit .NET 1.1)
System.Data.SqlServerCe	Compact Framework
System.Data.SqlTypes	Datentypen für SQL Server

Verbindungsorientierter Zugriff

Architektur

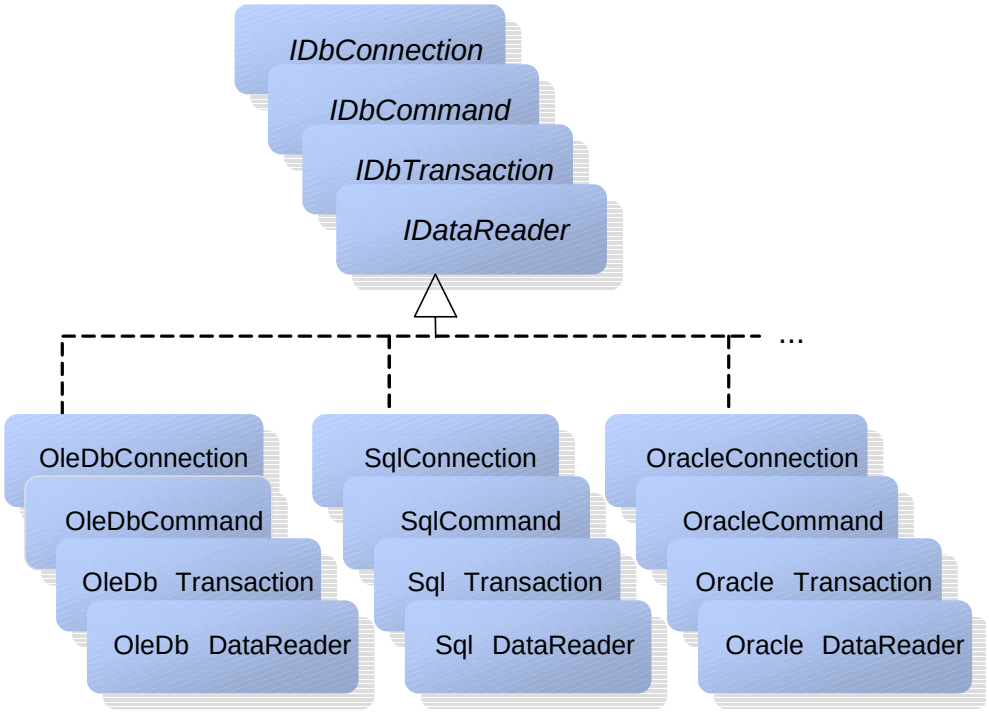
- DbConnection
  - repräsentiert Verbindung zu einer Datenquelle
- DbCommand
  - repräsentiert SQL-Kommando
- DbTransaction
  - repräsentiert Transaktion
  - Commands können innerhalb Transaktion ausgeführt werden
- DataReader
  - Ergebnis einer Datenbankabfrage
  - erlaubt sequentielles Lesen der Zeilen



Klassenhierarchie

- allgemeine Interface-Definitionen
  - IDbConnection
  - IDbCommand
  - IDbTransaction
  - IDataReader

- spezielle Implementierungen
  - OleDb: Implementierung für OLEDB
  - Sql: Implementierung für SQL Server
  - Oracle: Implementierung für Oracle
  - Odbc: Implementierung für ODBC
  - SqlCe: Implementierung für SQL Server CE-Datenbank



Programmmuster für verbindungsorientierten Datenzugriff

```
1.) Verbindung deklarieren;
try {
    1.) Verbindung zur Datenquelle anfordern

    2.) SQL-Kommandos ausführen

    3.) Ergebnis verarbeiten

    4.) Ressourcen freigeben
} catch ( Exception ) {
    Exception behandeln
} finally {
    try { // optional
        5.) Verbindung schliessen
    } catch (Exception) { Exception behandeln }
}
```

Beispiel: EmployeeReader(1)

```
using System;
using System.Data;
using System.Data.OleDb;

public class EmployeeReader {
   public static void Main() {
```

- Verbindung aufbauen

```
      string connStr = "provider=SQLOLEDB; data source=(local)\\NetSDK; " +
                       "initial catalog=Northwind; user id=sa; password=; ";
      IDbConnection con = null;              // Verbindung deklarieren
      try {
         con = new OleDbConnection(connStr);          //Verbindung erzeugen
         con.Open();                                  //Verbindung anfordern
```

- Kommandos ausführen

```
      //----- SQL-Kommando aufbauen
      IDbCommand cmd = con.CreateCommand();
      cmd.CommandText = "SELECT EmployeeID, LastName, FirstName FROM Employees";
      //----- SQL-Kommando ausführen; liefert einen OleDbDataReader
      IDataReader reader = cmd.ExecuteReader();
```

// Fortsetzung nächste Seite

Beispiel: EmployeeReader (2)

- Daten lesen und verarbeiten

```
        IDataReader reader = cmd.ExecuteReader();
        object[] dataRow = new object[reader.FieldCount];
        //----- Daten zeilenweise lesen und verarbeiten
        while (reader.Read()) { // solange noch Daten vorhanden sind
           int cols = reader.GetValues(dataRow); // tatsächliches Lesen
           for (int i = 0; i < cols; i++) Console.Write("| {0} " , dataRow[i]);
              Console.WriteLine();
        }
```

- Verbindung schliessen

```
     //----- Reader schliessen
     reader.Close();
     } catch (Exception e) {
        Console.WriteLine(e.Message);
     } finally {
        try {
           if (con != null)
              // Verbindung schliessen
              con.Close();
        } catch (Exception ex) { Console.WriteLine(ex.Message); }
     }
  }
}
```

Interface IDbConnection

- ConnectionString definiert Datenbankverbindung

```
string ConnectionString {get; set;}
```

- Öffnen und Schliessen der Verbindung

```
void Close();
void Open();
```

- Eigenschaften der Verbindung

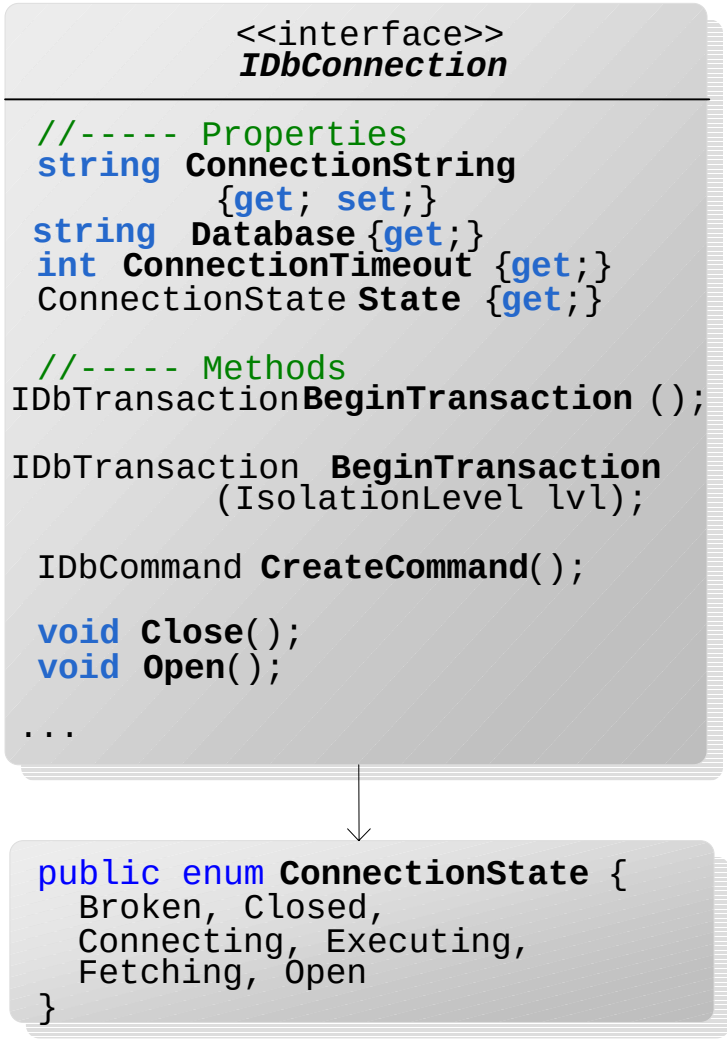
```
string  Database {get;}
int ConnectionTimeout {get;}
ConnectionState State {get;}
```

- Erzeugen eines Command-Objekts

```
IDbCommand CreateCommand();
```

- Erzeugen eines Transaction-Objekts

```
IDbTransaction BeginTransaction();
IDbTransaction
BeginTransaction(IsolationLevel lvl);
```



IDbConnection: Property ConnectionString

- Schlüssel-Wert-Paare durch Strichpunkt (;) getrennt
- Konfiguration der Verbindung
  - Name des Providers
  - Bezeichnung der Datenquelle
  - Authentifizierung des Benutzers
  - weitere Datenbank spezifische Einstellungen
- z.B. OLEDB: **`OleDbConnection`**

```
"provider=Microsoft.Jet.OLEDB.4.0;data source=c:\bin\LocalAccess40.mdb;"

"provider=SQLOLEDB; data source=127.0.0.1\\NetSDK;
    initial catalog=Northwind; user id=sa; password=; "

"provider=MSDAORA; data source=ORACLE8i7; user id=OLEDB; password=OLEDB;"
```

- z.B. MS-SQL-Server:**`SqlConnection`**

```
"data source=(local)\\NetSDK; initial catalog=Northwind; user id=sa;
    pooling=false; Integrated Security=SSPI; connection timout=20;"
```

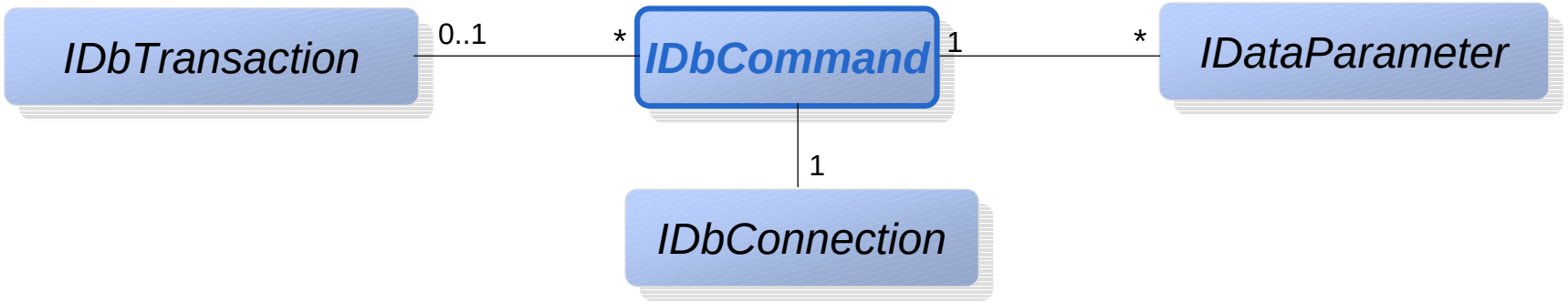
- z.B. ODBC: **`OdbcConnection`**

```
"dsn=ALG_TSAMPLE;UID=;PWD=;"
```

Data Provider

- Data Provider
- SQL Server .NET Data Provider (System.Data.SQL)
  - SQL Server 2000, SQL Server 7.0, MSDE
  - Verwendet eigenes Kommunikationsprotokoll (TDS)
- OLEDB .NET Data Provider (System.Data.OleDB)
  - SQLOLEDB: OLE DB Provider für SQL Server
  - MSDAORA: OLE DB Provider für Oracle
  - Microsoft.Jet.OLEDB 4.0: OLE DB für Microsoft Jet (Access)
- ODBC.NET Data Provider (System.Data.Odbc)
  - für alle DB mit ODBC Schnittstelle
- ODP.NET for Oracle (Oracle.DataAccess.Client)

Command-Objekte



- Command-Objekte definieren SQL-Anweisung oder Stored-Procedures
- werden für eine Connection ausgeführt
- können Parameter haben
- können zu einer Transaktion gehören

Interface IDbCommand

- CommandText definiert SQL-Anweisung oder Stored-Procedure

```
string CommandText {get; set;}
```

- Connection-Objekt

```
IDbConnection Connection {get; set;}
```

- Type- und Timeout-Eigenschaften

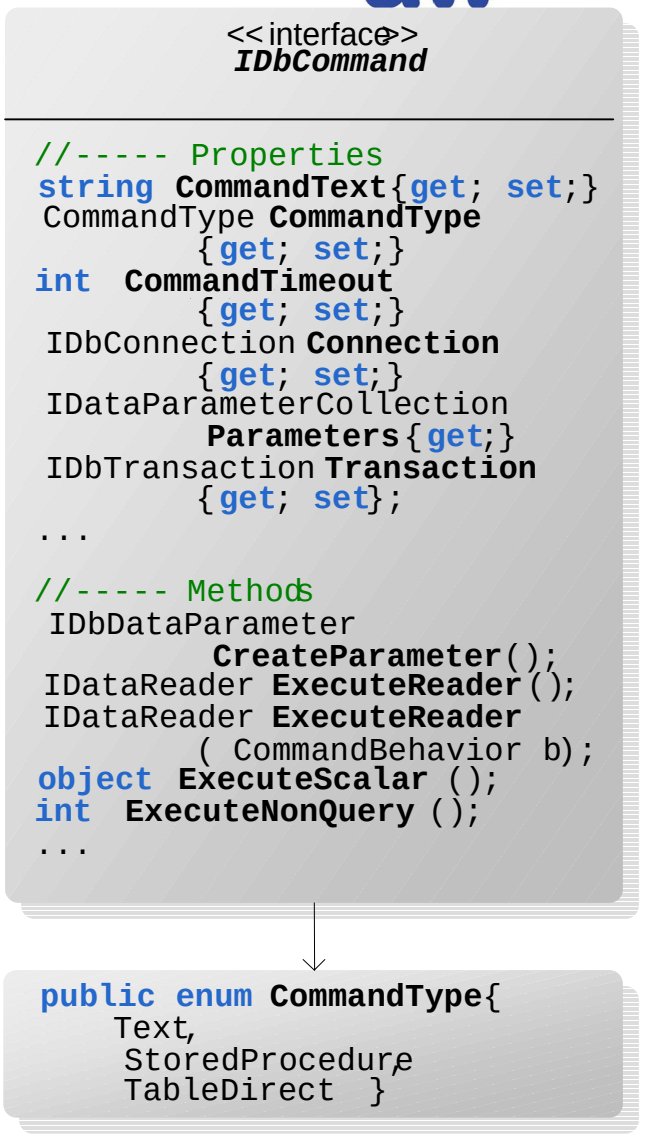
```
CommandType CommandType {get; set;}
int CommandTimeout {get; set;}
```

- Erzeugen und Zugriff auf Parameter

```
IDbDataParameter CreateParameter();
IDataParameterCollection Parameters {get;}
```

- Ausführen des Kommandos

```
IDataReader ExecuteReader();
IDataReader ExecuteReader(CommandBehavior b);
object ExecuteScalar();
int ExecuteNonQuery();
```



ExecuteReader-Methoden

```
IDataReader ExecuteReader()
IDataReader ExecuteReader( CommandBehavior behavior );
```

```
public enum CommandBehavior {
   CloseConnection, Default, KeyInfo, SchemaOnly,
   SequentialAccess, SingleResult, SingleRow
}
```

- Führt die in CommandText spezifizierte Datenbankabfrage (Query) aus
- Ergebnis ist IDataReader-Objekt
- Falls Kollonen-name mit SQL Schlüsselwort kollidiert: "[" "]" z.B. Select "[Start]"

Beispiel:

```
cmd.CommandText =
        "SELECT EmployeeID, LastName, FirstName FROM Employees ";
IDataReader reader = cmd.ExecuteReader();
```

ExecuteNonQuery-Methode

```
int ExecuteNonQuery();
```

- Führt die in CommandText spezifizierte Non-Query-Datenbankanweisung aus
  - UPDATE
  - INSERT
  - DELETE
  - CREATE TABLE
  - ...
- Rückgabewert ist die Anzahl der betroffenen Zeilen

Beispiel:

```
cmd.CommandText = "UPDATE Empls SET City = 'Seattle' WHERE iD=8";
int affectedRows = cmd.ExecuteNonQuery();
```

ExecuteScalar-Methode

```
object ExecuteScalar();
```

- Ergebnis ist Wert der 1. Spalte und der 1. Zeile der Datenbankabfrage
- CommandText ist typischerweise Aggregatsfunktion

Beispiel:

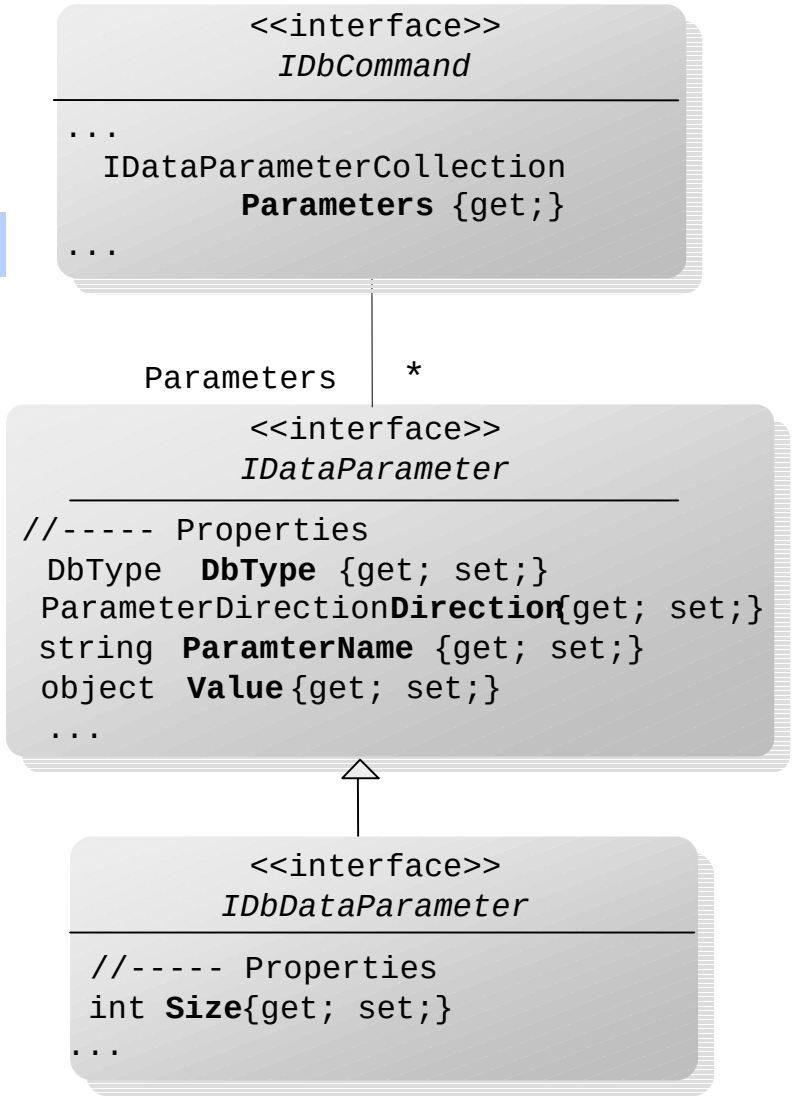
```
cmd.CommandText = "SELECT count(*) FROM Employees";
int count = (int) cmd.ExecuteScalar();
```

Parameter

- Command-Objekte erlauben Ein- und Ausgabeparameter

```
IDataParameterCollection Parameters {get;}
```

- Parameterobjekte spezifizieren
  - Name: Name des Parameters
  - Value: Wert des Parameters
  - DbDataType: Datentyp des Parameters
  - Direction: Richtung des Parameters
    - *Input*
    - *Output*
    - *InputOutput*
    - *ReturnValue*



Arbeiten mit Parametern

1. SQL-Kommando mit Platzhalter definieren

   OLEDB: Identifikation der Parameter über Position (Notation "?")

```
OleDbCommand cmd = new OleDbCommand();
cmd.CommandText = "DELETE FROM Empls WHERE EmployeeID = ?";
```

   SQL Server: Identifikation der Parameter über Namen (Notation "*@name*")

```
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "DELETE FROM Empls WHERE EmployeeID = @ID";
```

2. Parameter erzeugen und anfügen

```
cmd.Parameters.Add( new OleDbParameter("@ID", OleDbType.BigInt));
```

3. Werte festlegen und Kommando ausführen

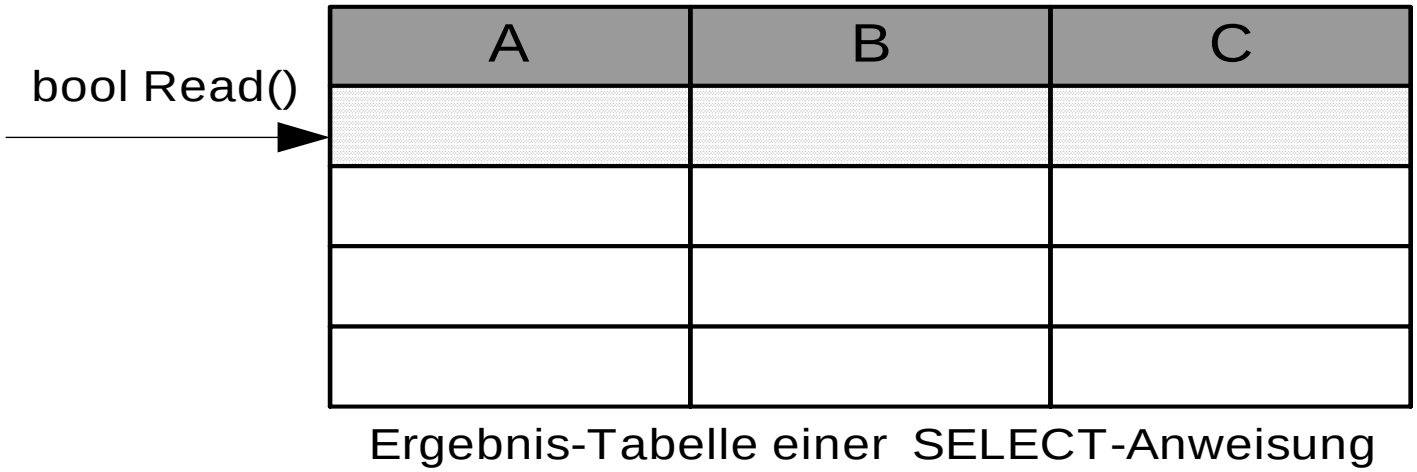
```
((IDataParameter)cmd.Parameters["@ID"]).Value = 1234;
cmd.ExecuteNonQuery();
```

DataReader

- Methode ExecuteReader() liefert DataReader-Objekt

```
IDataReader ExecuteReader()
IDataReader ExecuteReader( CommandBehavior behavior );
```

- DataReader ermöglicht das zeilenweise, sequentielle Lesen



Interface IDataReader

- Read liest nächste Zeile

```
bool Read();
```

- Zugriff auf Spaltenwerte mit Indexer

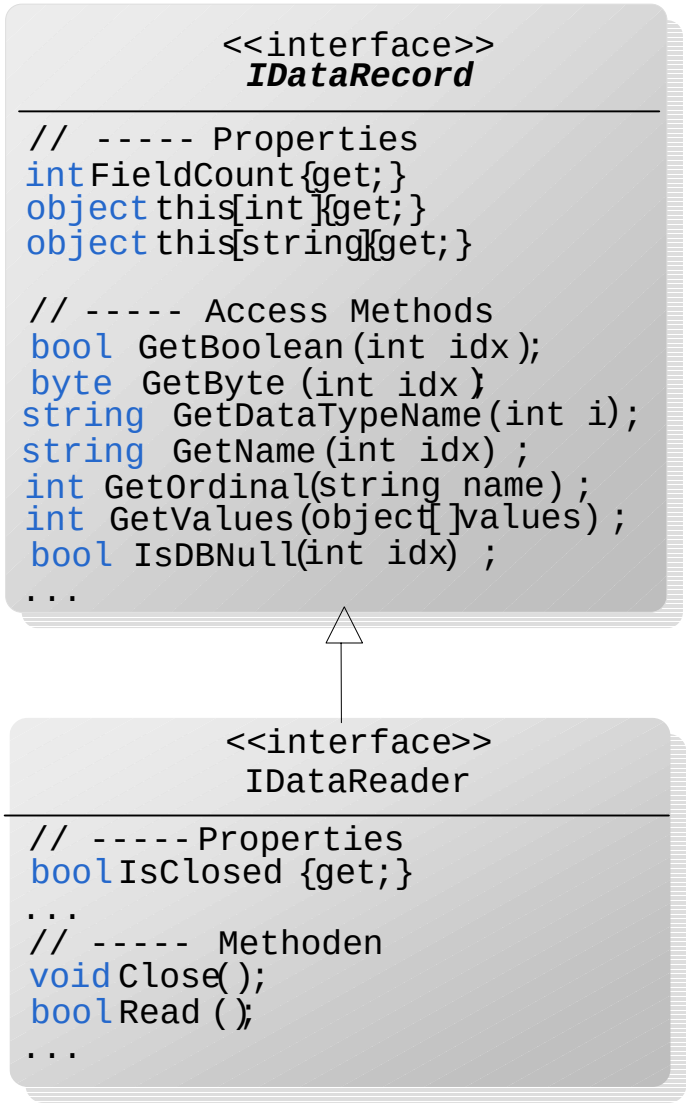
```
object this[int] {get;}
object this[string] {get;}
```

- Zugriffsmethoden mit Typumwandlung

```
bool GetBoolean(int idx);
byte GetByte(int idx);
...
```

- Zugriff auf Metainformation

```
string GetDataTypeName(int i);
string GetName(int idx);
int GetOrdinal(string name);
...
```



Arbeiten mit IDataReader

DataReader erzeugen und zeilenweise zugreifen

```
IDataReader reader = cmd.ExecuteReader();
while (reader.Read()) {
```

Spaltenwerte in object-Array auslesen

```
    object[ ] dataRow = new object[reader.FieldCount];
    int cols = reader.GetValues(dataRow);
```

oder Spaltenwerte einzeln durch Indexer auslesen

```
    object val0 = reader[0];
    object nameVal = reader["LastName"];
```

oder Spaltenwert mit typsicherer Zugriffsmethode auslesen

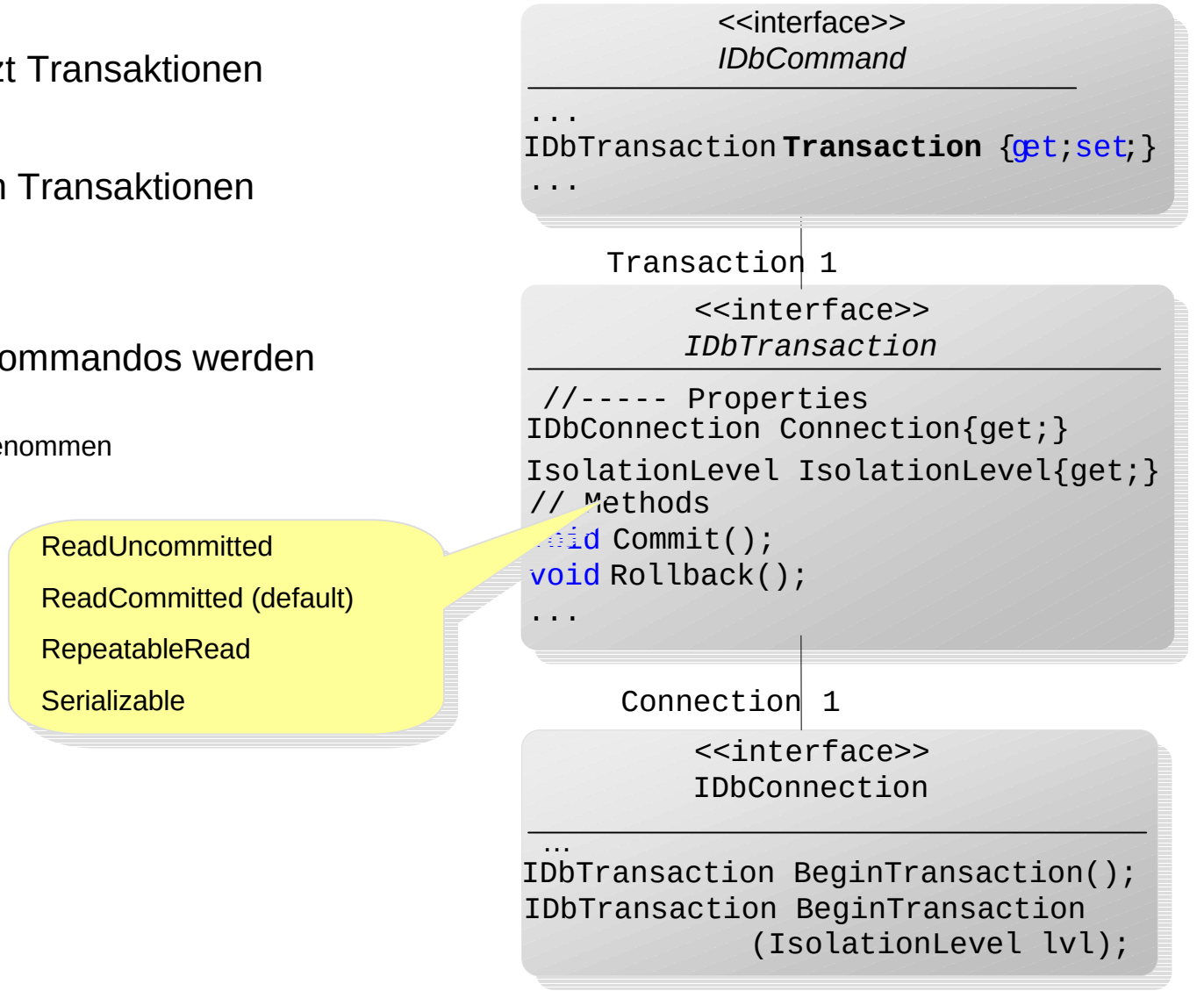
```
    string firstName = reader.getString(2);
```

DataReader schliessen

```
}
reader.Close();
```

Transaktionen

- ADO.NET unterstützt Transaktionen
- Kommandos werden Transaktionen zugeordnet
- Ausführungen der Kommandos werden
  - mit Commit bestätigt
  - mit Rollback zurückgenommen



Arbeiten mit Transaktionen (1)

1. Verbindung definieren und Transaction-Objekt erzeugen

```
SqlConnection con = new SqlConnection(connStr);
IDbTranaction trans = null;
try {
    con.Open();
    trans = con.BeginTransaction(IsolationLevel.ReadCommitted);
```

2. Command-Objekte erzeugen, dem Transaction-Objekt zuordnen und ausführen

```
    IDbCommand cmd1 = con.CreateCommand();
    cmd1.CommandText = "DELETE [OrderDetails] WHERE OrderId = 10258";
    cmd1.Transaction = trans;
    cmd1.ExecuteNonQuery();

    IDbCommand cmd2 = con.CreateCommand();
    cmd2.CommandText = "DELETE Orders WHERE OrderId = 10258";
    cmd2.Transaction = trans;
    cmd2.ExecuteNonQuery();
```

Arbeiten mit Transaktionen (2)

3. Transaktion bestätigen oder zurück nehmen

```
    trans.Commit();
catch (Exception e) {
    if (trans != null)
        trans.Rollback();
} finally {
    try {
            con.Close();
    }
}
```

Datenbankeigenschaften

- Daten werden von mehreren Benutzern gemeinsam verwendet.
- Jeder Benutzer kann das System benutzen, als ob er der einzige wäre.

- Spezielle Techniken sind notwendig um zu verhindern, dass sich mehrere Benutzer gegenseitig stören (*Concurrency Control)*.

Eigenschaften (ACID-Eigenschaften von Datenbanken)

1. **A**tomarität
2. Konsistenz (**C**onsistency) (Serialisierbarkeit)
3. **I**solation
4. **D**auerhaftigkeit

Eigenschaft 1: Atomarität

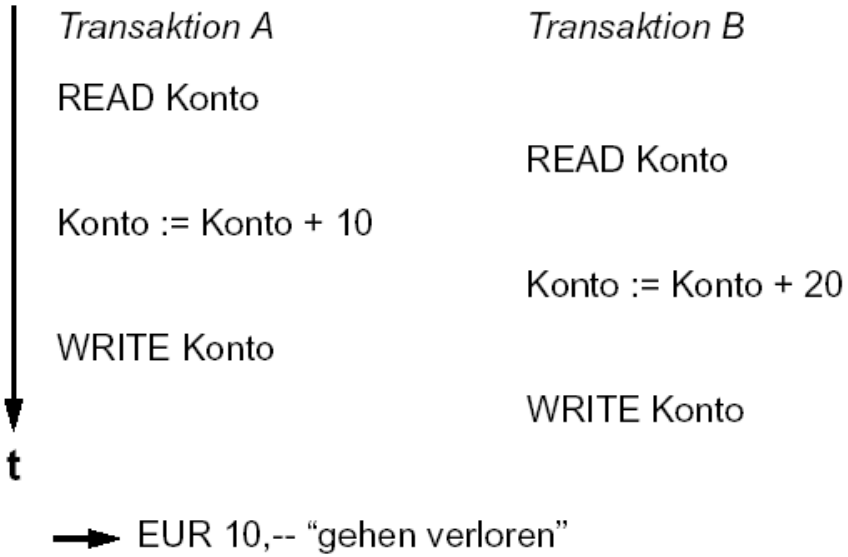
- eine Transaktion wird entweder zur Gänze ausgeführt oder gar nicht.

Beispiel: Guthaben bei einem Reisebüro soll auf ein Bankkonto umgebucht werden.

Kommt es zu einem Fehler, so wird weder die Abbuchung vom Reisebürokonto noch die Aufbuchung aufs Bankkonto durchgeführt.

Eigenschaft 2: Serialisierbarkeit

- Traditionelles Korrektheitskriterium für Parallelausführung von Transaktionen ist die ***Serialisierbarkeit***.
- Serialisierbarkeit garantiert: Das Ergebnis einer verzahnten Ausführung mehrerer Transaktionen entspricht dem Ergebnis irgendeiner Hintereinanderausführung, d.h. seriellen Ausführung, dieser Transaktionen.



Wir haben die 25 Mio nie verloren, Chef!
Das war nur ein Fehler im Computerausdruck.

Eigenschaft 3: Dauerhaftigkeit

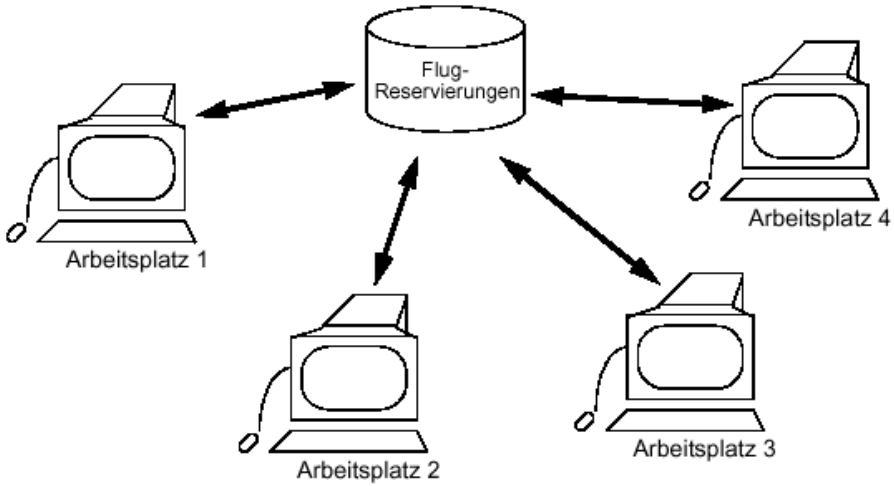
- Die Effekte einer einmal erfolgreich abgeschlossenen Transaktion bleiben auch bei etwaigen Hard- und Softwarefehlern in der Datenbank erhalten.
- Wird durch die Wiederanlauffähigkeit des Datenbanksystems gewährleistet.
- Standardverfahren:
  - Logging (Protokollierung der Änderungen)

Eigenschaft 4: Isolation

- Effekte einer Transaktion werden erst nach ihrem erfolgreichen Abschluss für andere Transaktionen sichtbar.

- Dient zur Vermeidung des *cascading rollback*. Sieht eine Transaktion Teilergebnisse einer anderen Transaktion, die abgebrochen wird, zieht das auch den Abbruch der ersteren mit sich.

Flugreservierungssystem:



Effekte bei nicht vollständiger Isolation

Werden Transaktionen nicht oder nicht vollständig voneinander isoliert, so können folgende Phänomene auftreten:

- Dirty Read

  Daten die noch nicht committed sind werden gelesen

- Unrepeatable Read

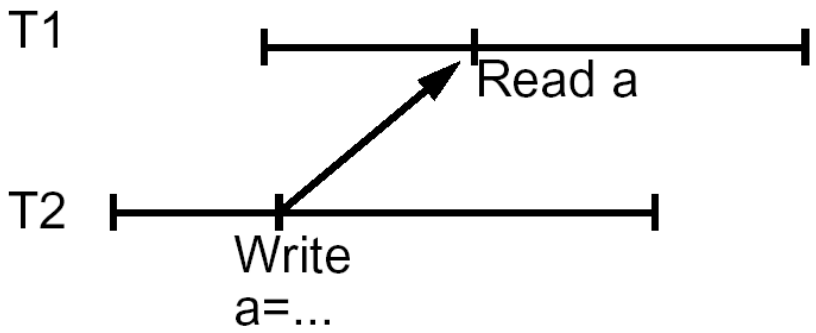
  2 Lese-Operationen in derselben Transaktion liefern unterschiedliche Werte eines Datensatzes

- Phantom Read

  Zusätzliche Datensätze erscheinen in späterer Leseoperation

Phänomen: Dirty Read

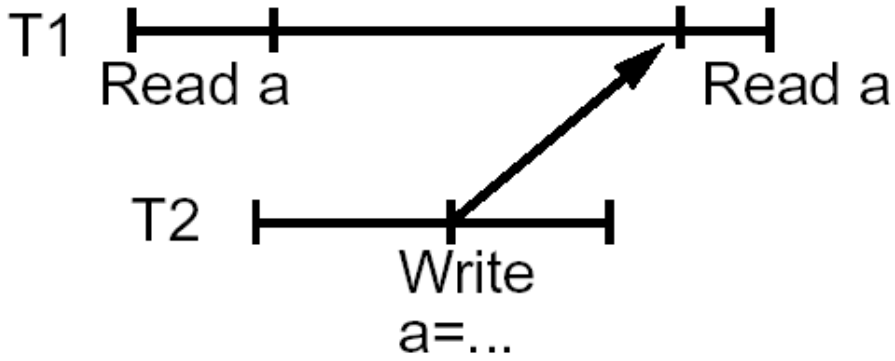
- T1 liest einen von T2 veränderten Wert,
  bevor T2 mit COMMIT abgeschlossen wurde.



Phänomen: Nonrepeatable Read

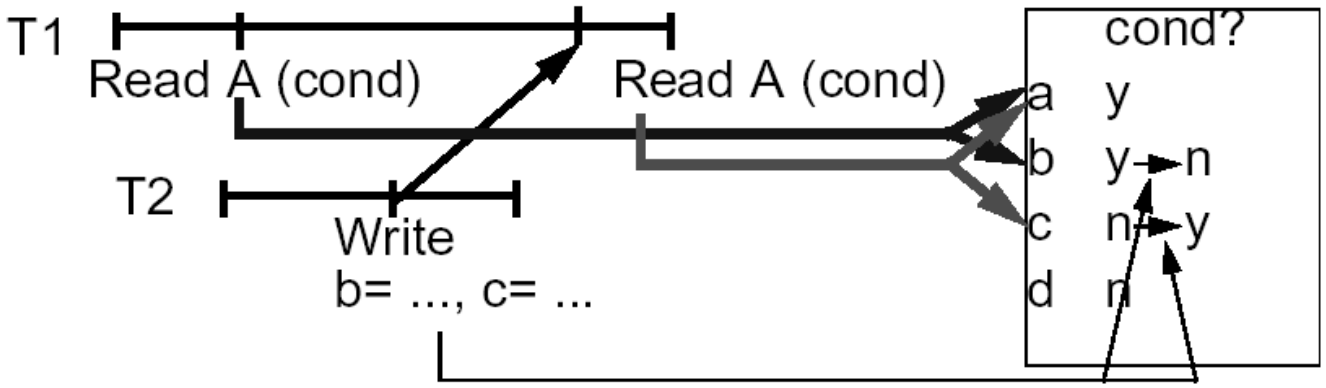
- T1 liest einen Satz.
- Zu einem späteren Zeitpunkt wird derselbe Satz von T1 noch einmal gelesen.

- Zwischen den beiden Leseoperationen von T1 hat T2 den Wert dieses Satzes verändert. T1 bekommt also zwei verschiedene Werte zurückgeliefert.



Phänomen: Phantom Read

- T1 liest eine Menge von Sätzen aufgrund bestimmter Kriterien.
- Dieser Lesevorgang wird später von T1 wiederholt.
- Zwischen den beiden Leseoperationen ändert T2 einige Werte, sodass beim 2. Lesevorgang eine andere Menge von Sätzen die Kriterien erfüllt und an T1 zurückgeliefert wird.



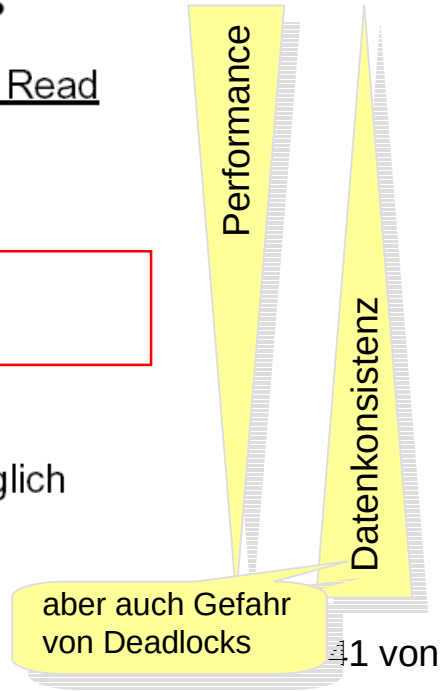
Isolation Levels

- Isolationsstufen definieren Verwendung von Lese- und Schreibsperren bei Transaktionen
- Isolation Level bestimmen Read/Write-Locks die gesetzt werden

```
public enum IsolationLevel {
   ReadUncommitted, ReadCommitted, RepeatableRead, Serializable
}
```

Vermeidung der Phänomene durch *Isolation levels*

Level	Dirty Read	Nonrep. Read	Phantom Read
READ UNCOMMITTED	möglich	möglich	möglich
READ COMMITTED	nicht möglich	möglich	möglich
REPEATABLE READ	nicht möglich	nicht möglich	möglich
SERIALIZABLE	nicht möglich	nicht möglich	nicht möglich



Locking

- Grundsatz: ACID is good, but take it in short doses
- **Pessimistic Locking**
  - DB-Sperre wird gesetzt
  - Datensatz wird gelesen
  - Datensatz wird von Benutzer bearbeitet
  - Datensatz wird geschrieben
  - DB-Sperre wird aufgehoben
- **Optimistic Locking**
  - Datensatz wird gelesen
  - **Kopie** von Datensatz wird von Benutzer bearbeitet
  - DB-Sperre wird gesetzt
  - Originaldaten (oder Zeitstempel) werden mit DB Daten verglichen
    -> unterschiedlich ? -> Fehler, Abbruch
  - veränderte Kopie von Datensatz wird geschrieben
  - DB-Sperre wird aufgehoben

Verbindungsloser Zugriff

Motivation und Idee

- viele parallele, teils lang andauernde Datenzugriffe unterstützen
- verbindungsorientierte Datenzugriffe zu aufwendig

Idee

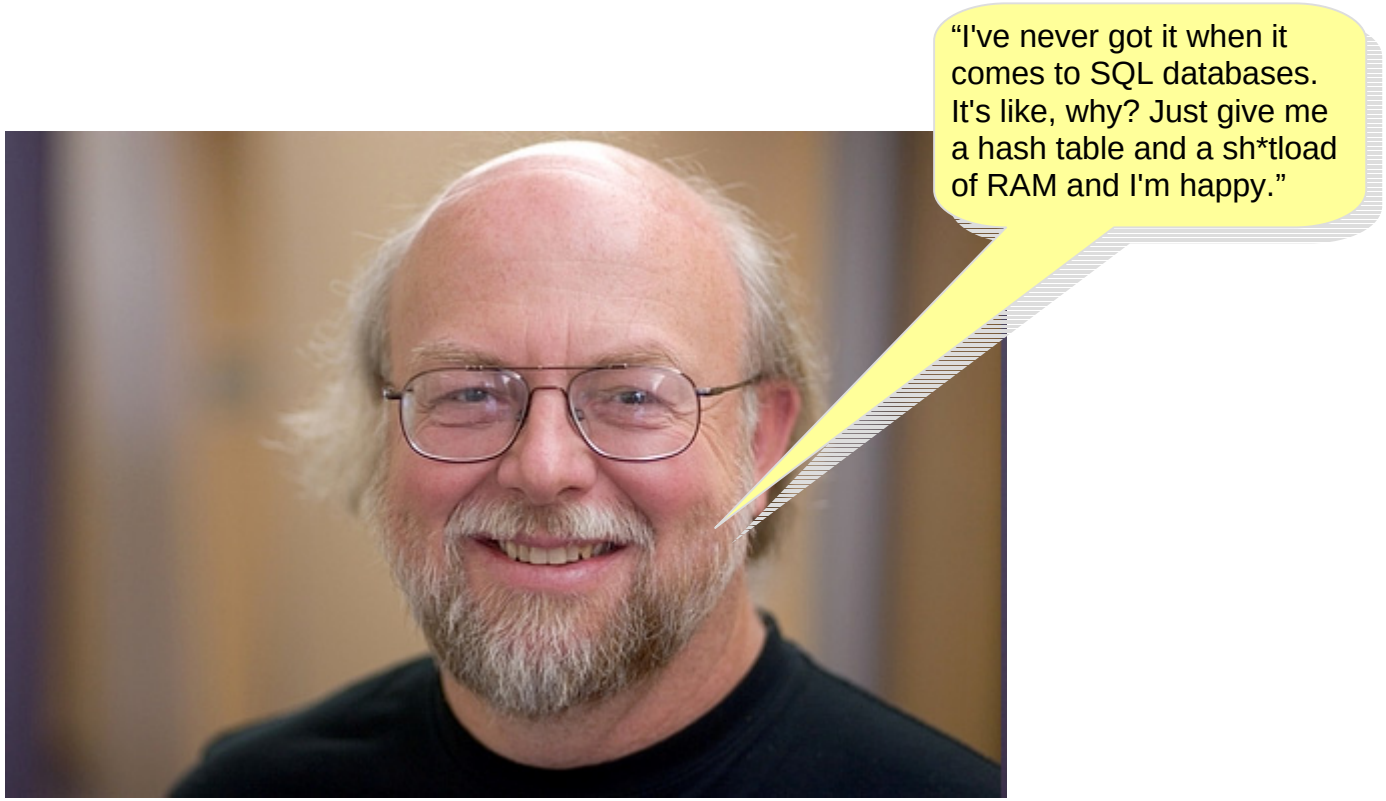
- Daten im Hauptspeicher zwischengespeichert

  -> „Hauptspeicher-Datenbank“
- Verbindung zur Datenquelle nur kurzzeitig für Lesen und Updates

  -> DataAdapter
- Hauptspeicherdatenbank unabhängig von Datenquelle

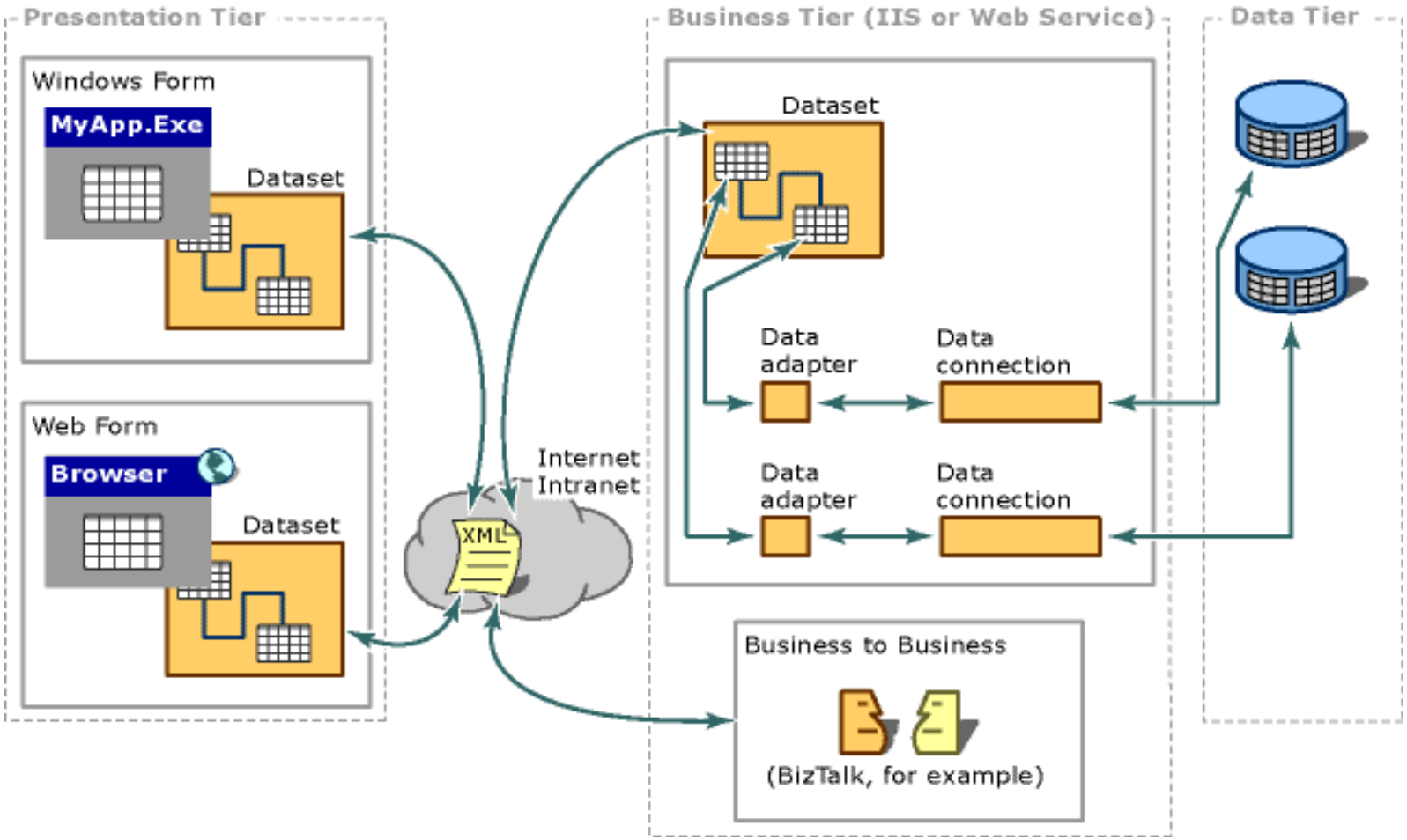
  -> gegengleiche Änderungen sind möglich

In Memory Databases



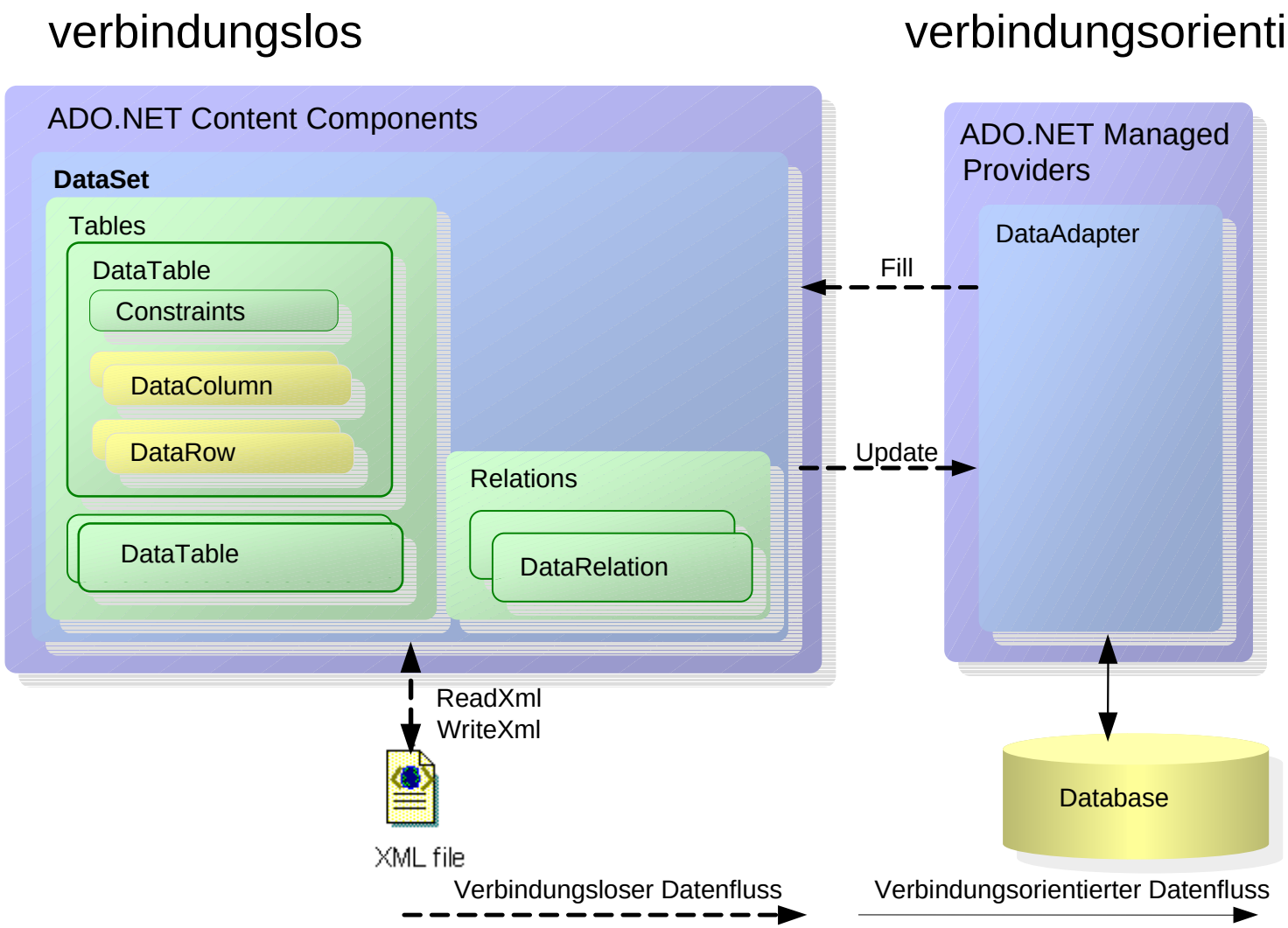
James Gosling, the inventor of Java

Microsoft 3-Tier Architektur



entnommen aus: Introduction to Data Access with ADO.NET, http://msdn.microsoft.com/library/

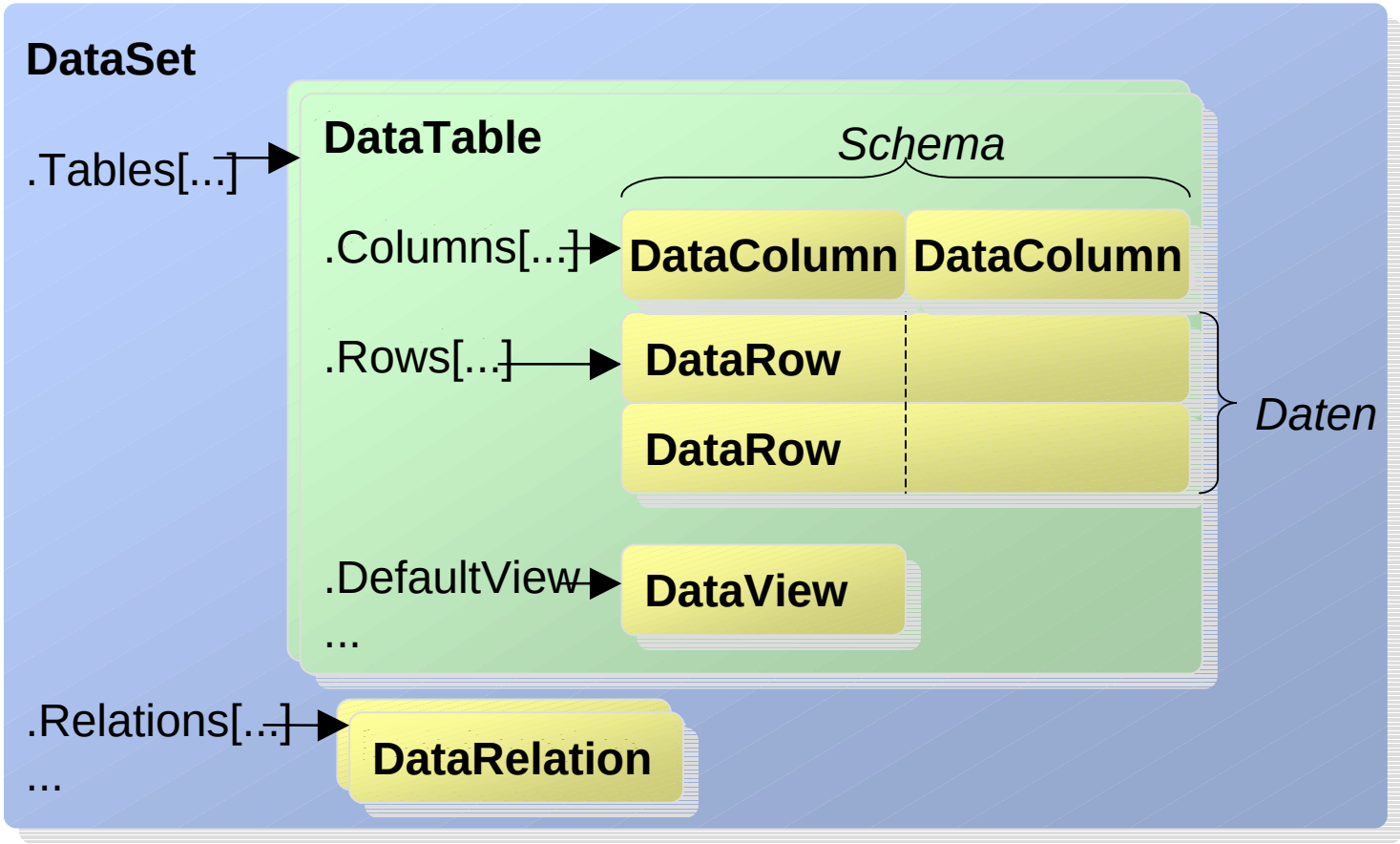
Architektur verbindungsloser Zugriff



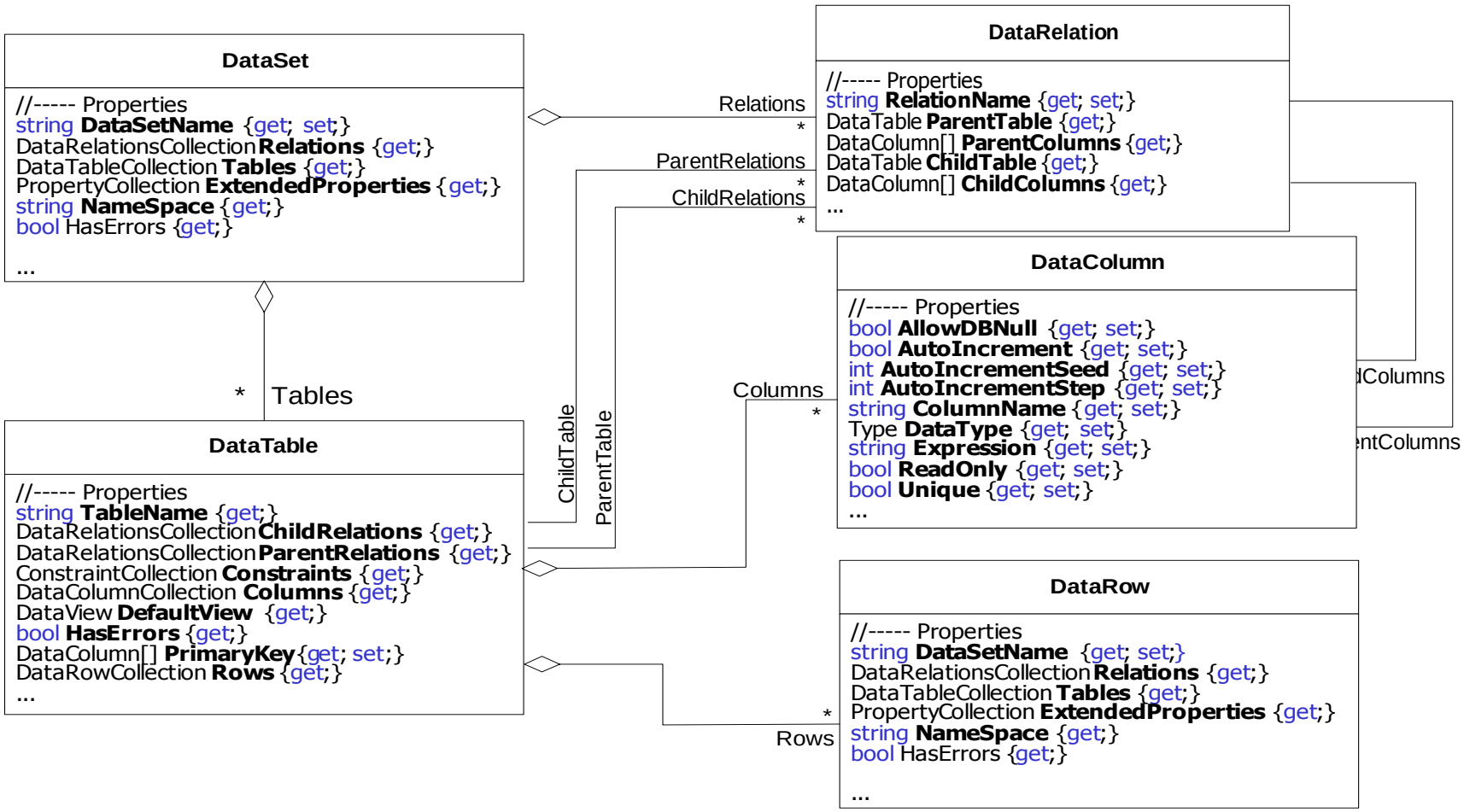
DataSet

- Hauptspeicher-Datenbank
  - relationale Struktur
  - objektorientierte Schnittstelle
- DataSet besteht aus
  - Sammlung von DataTables
  - Sammlung von DataRelations
- DataTables bestehen aus
  - Sammlung von DataTableColumns (= Schemadefinition)
  - Sammlung von DataTableRows (= Daten)
  - DefaultView (DataTableView siehe später)
- DataRelations
  - verknüpften zwei DataTable-Objekte über DataColumn-Objekte
  - definieren ParentTable und ParentColumns
    und ChildTable und ChildColumns

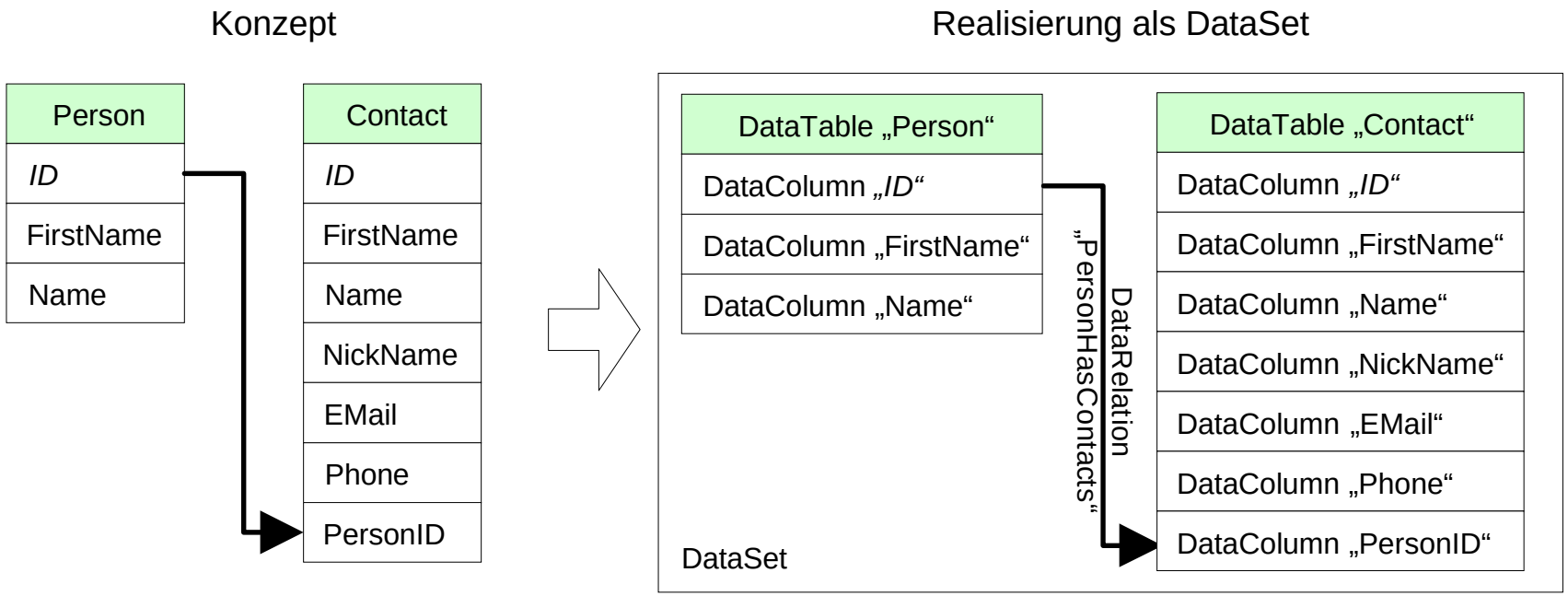
DataSet Aufbau



DataSet Klassendiagramm



Beispiel: Kontaktliste



Realisierungsschritte

- Schema definieren
- Daten definieren
- auf Daten zugreifen

Kontaktliste: Schema definieren (1)

DataSet "PersonContacts" und DataTable "Person“ anlegen

```
DataSet ds = new DataSet("PersonContacts");
DataTable personTable = new DataTable("Person");
```

Spalte "ID" definieren und Eigenschaften festlegen

```
DataColumn col = new DataColumn();
col.DataType = typeof(System.Int64);
col.ColumnName = "ID";
col.ReadOnly = true;
col.Unique = true;                  // values must be unique
col.AutoIncrement = true;          // keys are assigned automatically
col.AutoIncrementSeed = -1;        // first key starts with -1
col.AutoIncrementStep = -1;        // next key = prev. key - 1
```

Spalte zur Tabelle anfügen und als Primärschlüssel festlegen

```
personTable.Columns.Add(col);
personTable.PrimaryKey = new DataColumn[ ] { col };
```

Kontaktliste: Schema definieren (2)

Spalte "FirstName" definieren und anfügen

```
col = new DataColumn();
col.DataType = typeof(string);
col.ColumnName = "FirstName";
personTable.Columns.Add(col);
```

Spalte "Name" definieren und anfügen

```
col = new DataColumn();
col.DataType = typeof(string);
col.ColumnName = "Name";
personTable.Columns.Add(col);
```

Tabelle zum DataSet hinzufügen

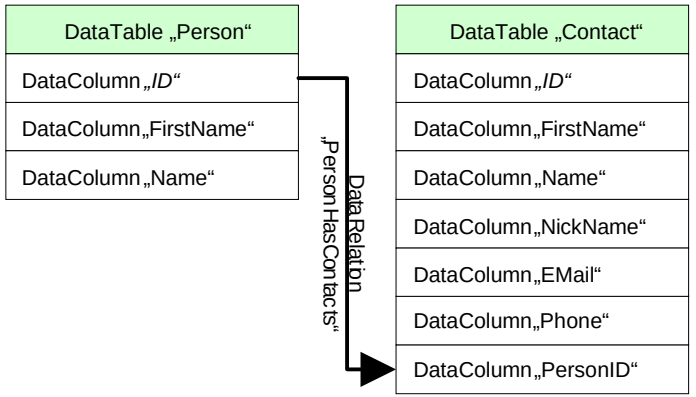
```
ds.Tables.Add(personTable);
```

Analog Tabelle "Contact" definieren

```
DataTable contactTable = new DataTable("Contact");
...
ds.Tables.Add(contactTable);
```

Kontaktliste: Relation definieren

Relation PersonHasContacts erzeugen
und zu DataSet hinzufügen




```
DataColumn parentCol = ds.Tables["Person"].Columns["ID"];
DataColumn childCol = ds.Tables["Contact"].Columns["PersonID"];

DataRelation rel = new DataRelation("PersonHasContacts", parentCol,
childCol);
ds.Relations.Add(rel);
```

Kontaktliste: Datensätze definieren

School of Engineering

Neue Zeile erzeugen und Spaltenwerte festlegen

```
DataRow personRow = personTable.NewRow();
personRow[1] = "Wolfgang";
personRow["Name"] = "Beer";
```

Zeile zur Tabelle "Person" hinzufügen

```
personTable.Rows.Add(personRow);
```

Neue Zeile zur Tabelle "Contact" hinzufügen

```
DataRow contactRow = contactTable.NewRow ();
contactRow[0] = "Wolfgang";
...
contactRow["PersonID"] = (long)personRow["ID"]; // defines relation
contactTable.Rows.Add (contactRow);
```

Alle Änderungen im DataSet übernehmen

```
ds.AcceptChanges();
```

Kontaktliste: Auf Daten zugreifen

über Zeilen in personTable iterieren und Name ausgeben

```
foreach (DataRow person in personTable.Rows) {
    Console.WriteLine("Contacts of {0}:", person["Name"]);
```

über Relation "PersonHasContacts" auf Kontakte zugreifen und alle Kontakte ausgeben

```
   foreach (DataRow contact in person.GetChildRows("PersonHasContacts")) {
      Console.WriteLine("{0}, {1}: {2}", contact[0], contact["Name"],
                                                     contact["Phone"] );
   }
```

DataSet: Änderungsmanagement

- DataSets protokollieren alle Änderungen
- Änderungen werden erst mit Aufruf von acceptChanges übernommen
- oder mit rejectChanges verworfen

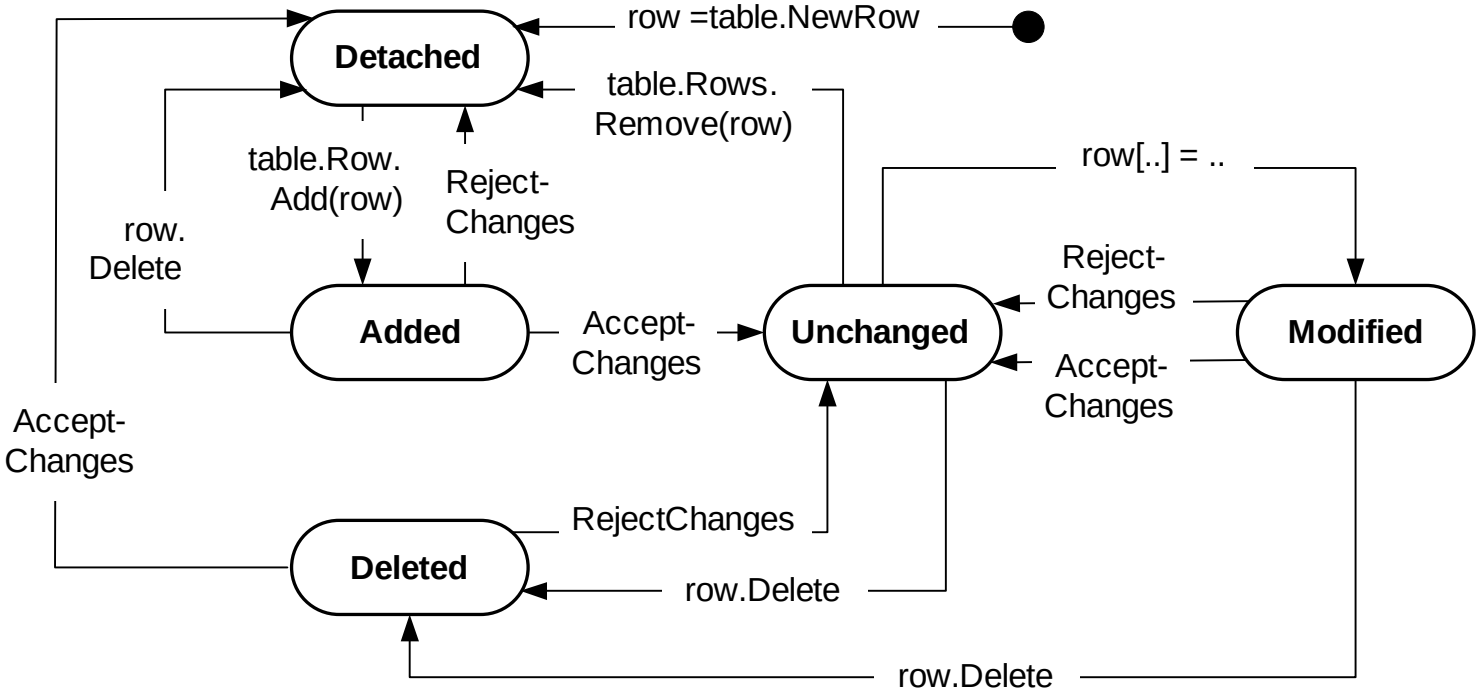
```
   ...
   if (ds.HasErrors) {
      ds.RejectChanges();
   } else {
      ds.AcceptChanges();
   }
}
```

Zustandsdiagramm eines DataRow-Objekts

- DataRow-Objekte haben unterschiedliche Zustände

```
public DataRowState RowState {get;}
```

```
public enum DataRowState {
   Added, Deleted, Detached, Modified, Unchanged
}
```



DataRowVersion

- DataSets speichern unterschiedliche Versionen von Datenzeilen:

```
public enum DataRowVersion {
   Current, Original, Proposed, Default
}
```

Current: aktuelle Werte

Original: ursprüngliche Werte

Proposed: vorgeschlagene Werte (Werte, die in Bearbeitung sind)

Default: Die Standardversion, basiert auf DataRowState-Wert

DataRowState	Default
Added, Modified, Unchanged	Current
Deleted	Original
Detached	Proposed

Beispiel:

```
bool hasOriginal = personRow.HasVersion(DataRowVersion.Original);
if (hasOriginal) {
   string originalName = personRow["Name", DataRowVersion.Original];
}
```

Exception Handling

School of Engineering

- ADO.NET überprüft Gültigkeit von Operationen auf DataSets
- und wirft DataExceptions

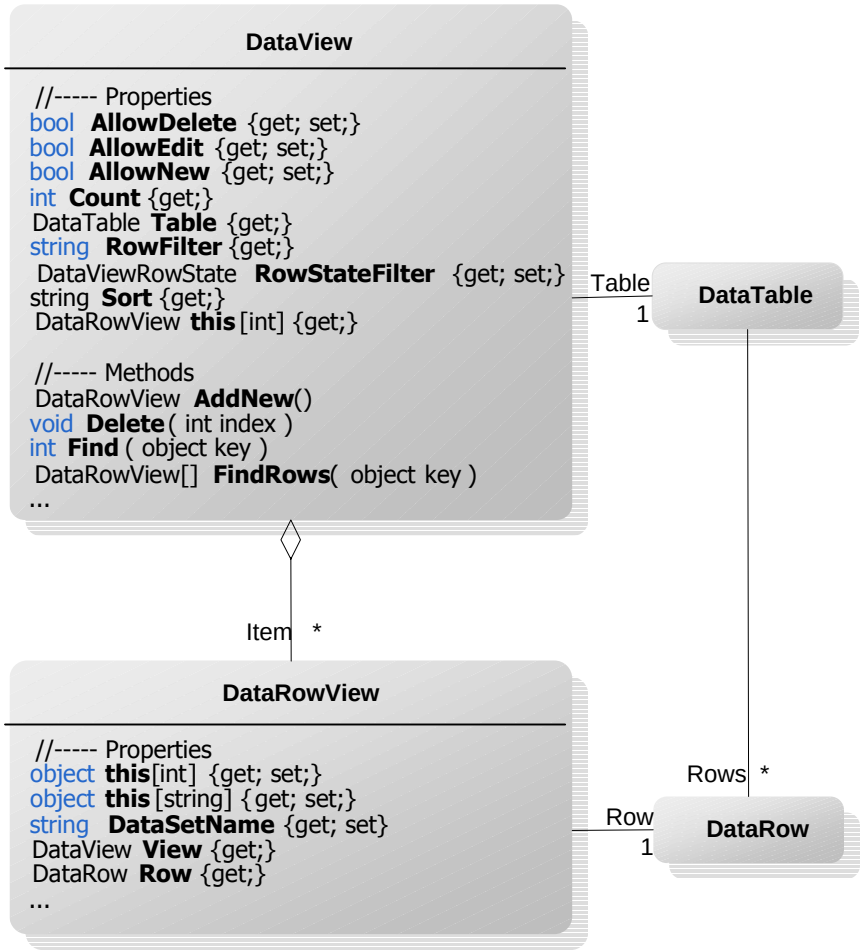
```
DataException
      ConstraintException
      DeletedRowInaccessibleExcception
      DuplicateNameException
      InvalidConstraintException
      InvalidExpressionException
      MissingPrimaryKeyException
      NoNullAllowedException
      ReadOnlyException
      RowNotInTableException
      ...
```

DataView

- DataView-Objekte für Sichten auf Tabellen
  - RowFilter: Filtern auf Basis eines Filter-Ausdrucks
  - RowStateFilter: Filtern auf Basis von RowStates
  - Sort: Sortieren der Zeilen nach Spalten

- DataViews erlauben
  - Ändern der Daten (# DB Views)
  - keine Verknüpfung von Tabellen
  - Schnelles Suchen (auf Basis der sortierten Spalten)

- DataView-Objekte können von GUI-Elementen dargestellt werden
  - z.B. DataGrid



Arbeiten mit DataView

School of Engineering

DataView erzeugen

```
DataView dataView = new DataView(personTable);
oder
DataView dataView =  dataSet1.Tables["Persons"].DefaultView;
```

Filter- und Sortierkriterium festlegen

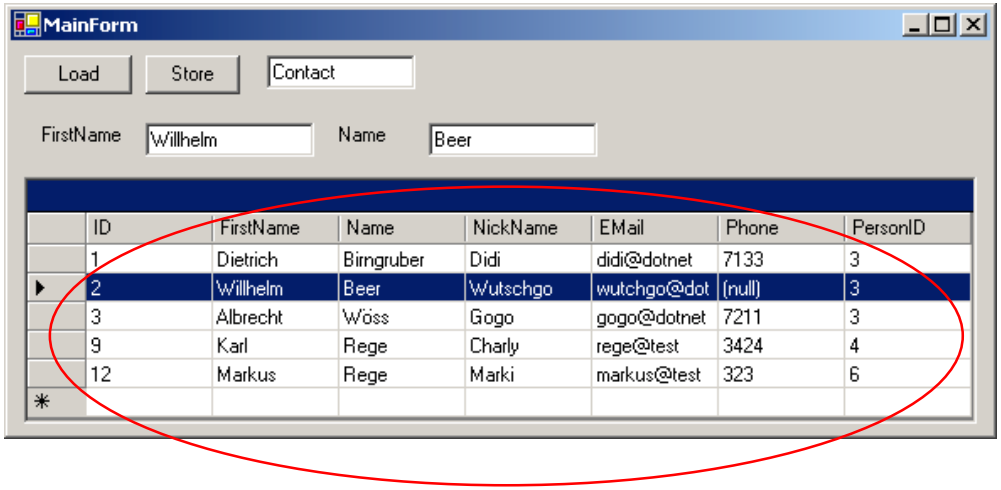
```
dataView.RowFilter = "FirstName >= 'K' AND FirstName <= 'T'";
dataView.Sort = "Name ASC";        // sort by Name in ascending order
```

Suchen und Auslesen der Werte

```
dataView.RowFilter = "Name = 'Beer'";
DataRowView drv = dataView[0];
Console.WriteLine(drv["Name"]);
```

DataGridView Control

- DataGridView
  - GUI Element zur Darstellung
    - *DataSet*
    - *DataView*
    - *andere*



Daten in DataGridView anzeigen

```
dataGridView1.DataSource =  dataSet1.Tables["Persons"].DefaultView;

dataGridView1.Columns["id"].Visible = false; // id-Kolonne Ausblenden
```

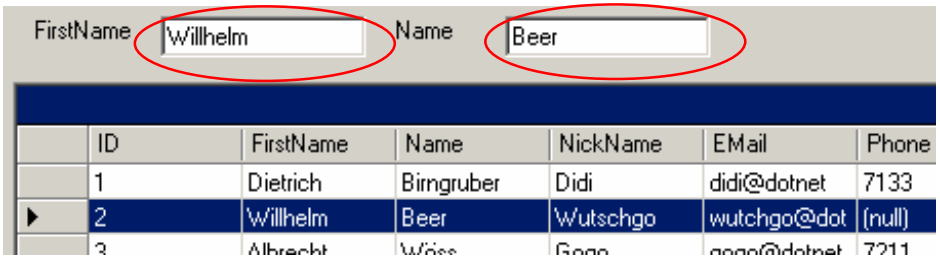
Sehr reichhaltiges aber auch komplexes Control

http://forums.mysql.com/read.php?38,115063,115063

http://www.akadia.com/services/dotnet_combobox_in_datagrid.html

DataBinding anderer Controls

- Andere WindowControls unterstützen ebenfalls DataBindings
- z.B. TextField, ComboBox, etc



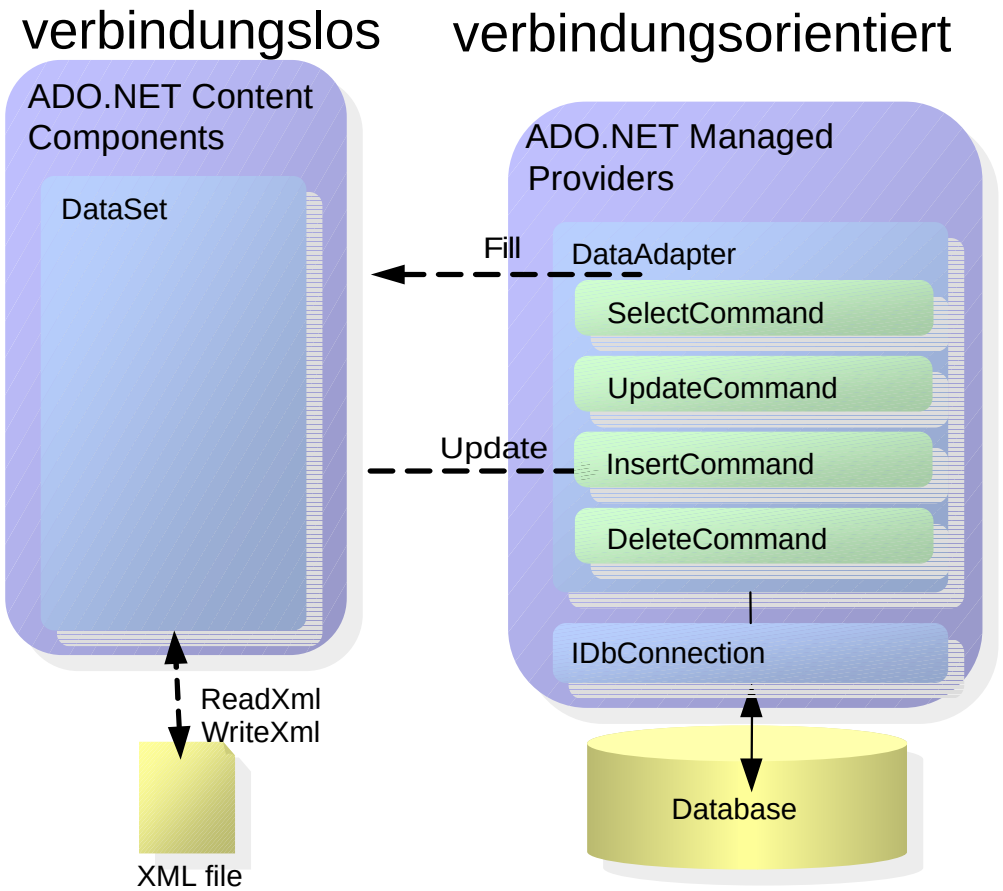
Anzeige von einzelnen Werten

```
textBox2.DataBindings.Add("Text",dataView ,"FirstName");
textBox3.DataBindings.Add("Text",dataView ,"Name");
comboBox1.DataSource = dataView ;
comboBox1.ValueMember = "Name";
comboBox1.DisplayMember = "Name";
```

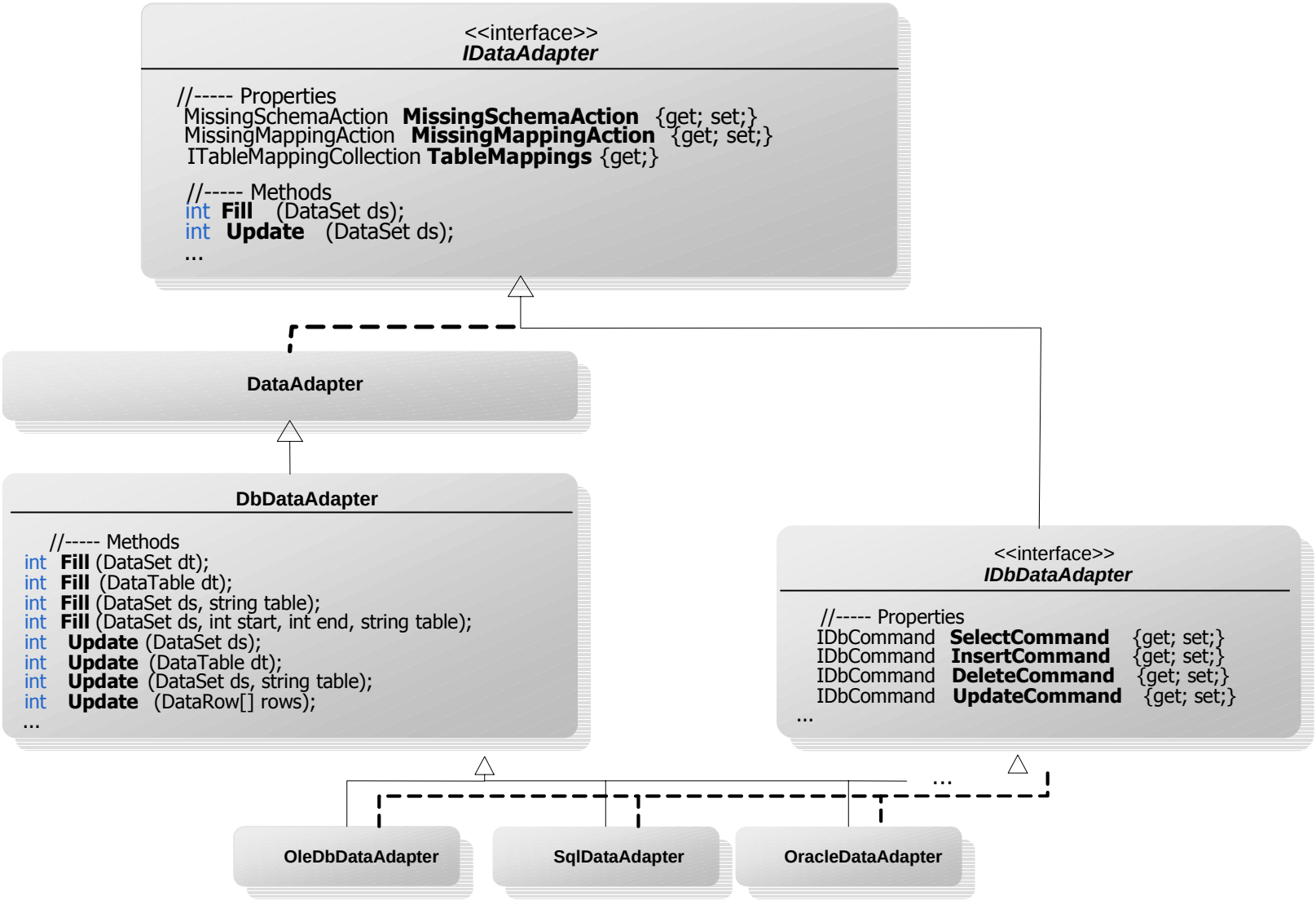
Datenbankverbindung mit DataAdapter

Architektur

- DataAdapter für Verbindung zur Datenquelle
  - Fill: Füllen des DataSets
  - Update: Änderungen zurück schreiben
- verwenden Command-Objekte
  - SelectCommand
  - InsertCommand
  - DeleteCommand
  - UpdateCommand



DataAdapter Klassendiagramm



DataAdapter: Laden von Daten

DataAdapter-Objekt erzeugen und SelectCommand setzen

```
IDbDataAdapter adapter = new OleDbDataAdapter();
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = new OleDbConnection ("provider=SQLOLEDB; ..." );
cmd.CommandText = "SELECT * FROM Contact";
adapter.SelectCommand = cmd;
```

Daten aus Datenquelle lesen und DataTable "Contact" füllen

```
((DbDataAdapter)adapter).Fill(ds, "Contact");
```

Änderungen akzeptieren oder bei Fehlern verwerfen
Adapter löschen

```
if (ds.HasErrors) ds.RejectChanges();
else ds.AcceptChanges();
if (adapter is IDisposable) ((IDisposable)adapter).Dispose();
```

DataAdapter: Laden von Schema und Daten

DataAdapter-Objekt erzeugen und SelectCommand setzen

```
IDbDataAdapter adapter = new OleDbDataAdapter();
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = new OleDbConnection ("provider=SQLOLEDB; ..." );
cmd.CommandText = "SELECT * FROM Contact";
adapter.SelectCommand = cmd;
```

Aktion bei fehlendem Schema festlegen und Zuordnung von Tabellen definieren

```
adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;
```

Daten aus Datenquelle lesen und DataTable "Contact" füllen

```
((DbDataAdapter)adapter).Fill(ds, "Contact");
```

Änderungen akzeptieren oder bei Fehlern verwerfen
Adapter löschen

```
if (ds.HasErrors) ds.RejectChanges();
else ds.AcceptChanges();
if (adapter is IDisposable) ((IDisposable)adapter).Dispose();
```

DataAdapter: Speichern von Änderungen (1)

School of Engineering

- Änderungen werden mit Update in die Datenquelle geschrieben
- Update-, Insert- und DeleteCommand definieren, wie Änderungen zu schreiben sind
- CommandBuilder kann Update-, Insert- und DeleteCommand aus SelectCommand automatisch erzeugen (in einfachen Fällen)
- Konfliktmanagement bei Updates
  - Vergleich von Daten in DataTable und Datenquelle
  - Bei Konflikt wird DBConcurrencyException ausgelöst

DataAdapter: Speichern von Änderungen (2)

DataAdapter mit SELECT-Ausdruck und Connection erzeugen

```
OleDbConnection con = new OleDbConnection ("provider=SQLOLEDB; …");
adapter = new OleDbDataAdapter("SELECT * FROM Contact", con);
```

Mit CommandBuilder-Objekt Updatekommandos automatisch erzeugen

```
OleDbCommandBuilder cmdBuilder = new OleDbCommandBuilder(adapter);
```

Update aufrufen und Konflikte behandeln

```
try {
   adapter.Update(ds,"Contact");
} catch (DBConcurrencyException) {
   // Handle the error, e.g. by reloading the DataSet
}
adapter.Dispose();
```

Integration mit XML

Integration DataSets und XML

- DataSets und XML sind stark integriert:
  - Serialisierung von DataSets als XML-Daten
  - XML-Dokumente als Datenquellen für DataSets
  - Schemas von DataSets aus XML-Schemas
  - *stark typisierte* DataSets aus XML-Schemas generiert
  - Zugriff auf DataSets auch über XML-DOM möglich

- Diese starke Integration wird bei verteilten Systemen, z.B. bei Web-Services, verwendet
  - (siehe *Microsoft 3-Tier Architektur*)

Zugriff auf DataSets über XML-DOM

School of Engineering

- XmlDataDocument erlaubt Zugriff über XML-DOM-Schnittstelle
- Synchronisation der Änderungen im XmlDataDocument und DataSet

Beispiel:

- XmlDataDocument-Objekt für DataSet erzeugen
- Daten im DataSet ändern

```
XmlDataDocument xmlDoc = new XmlDataDocument(ds);
...
DataTable table = ds.Tables["Person"];
table.Rows.Find(3)["Name"] = "Changed Name!";
```

- geänderte Daten über XmlDataDocument-Objekt auslesen

```
XmlElement root = xmlDoc.DocumentElement;
XmlNode person = root.SelectSingleNode("descendant::Person[ID='3']");
Console.WriteLine("Access via XML: \n" + person.OuterXml);
```

Schreiben und Lesen von XML-Daten

School of Engineering

- Methoden zum Schreiben und Lesen von XML-Daten

```
public class DataSet : MarshalByValueComponent, IListSource,
                       ISupportInitialize, ISerializable {
   public void WriteXml( Stream stream );
   public void WriteXml( string fileName );
   public void WriteXml( TextWriter writer);
   public void WriteXml( XmlWriter writer );
   public void WriteXml( Stream stream, XmlWriteMode m );
   public void ReadXml ( Stream stream );
   public void ReadXml ( string fileName );
   public void ReadXml ( TextWriter writer);
   public void ReadXml ( XmlWriter writer );
   public void ReadXml ( Stream stream, XmlReadMode m );
   ...
}
```

```
public enum XmlWriteMode {DiffGram, IgnoreSchema, WriteSchema}
```

```
public enum XmlReadMode {
   Auto, DiffGram, IgnoreSchema, ReadSchema, InferSchema, Fragment  }
```

Beispiel: Schreiben und Lesen von XML-Daten

School of Engineering

Schreiben von XML

```
ds.writeXML("personcontact.xml");
```

- Lesen von XML

  mit XmlReadMode.Auto wird automatisch Schema aufgebaut

```
DataSet ds = new DataSet();
ds.readXML("personcontact.xml",
           XmlReadMode.Auto);
```

```
<?xml version="1.0" standalone="yes" ?>
<PersonContacts>
-  <Person>
      <ID>1</ID>
      <FirstName>Wolfgang</FirstName>
      <Name>Beer</Name>
   </Person>
-  <Person>
      <ID>2</ID>
      <FirstName>Dietrich</FirstName>
      <Name>Birngruber</Name>
   </Person>
   <Contact>
      <ID>1</ID>
      <FirstName>Dietrich</FirstName>
      <Name>Birngruber</Name>
      <NickName>Didi</NickName>
      <EMail>didi@dotnet.jku.at</EMail>
      <Phone>7133</Phone>
      <PersonID>2</PersonID>
   </Contact>
-  <Contact>
      <ID>2</ID>
      <FirstName>Wolfgang</FirstName>
      <Name>Beer</Name>
      ...
      <PersonID>1</PersonID>
   </Contact>
  </PersonContacts>
```

DataSet und XML-Schema

School of Engineering

- DataSets erlauben auch Schreiben und Lesen von XML-Schemas
  - GetXmlSchema: Liefert das Schema des ds als String
  - WriteXmlSchema: Schreiben von XML-Schema
  - ReadXmlSchema: Lesen eines XML-Schema und Aufbau des DataSets
  - InferXmlSchema: Lesen von XML-Daten, um daraus ein Schema zu gewinnen

```
   ...
   public string GetXmlSchema();

   public void WriteXmlSchema ( Stream stream );
   public void WriteXmlSchema ( string fileName );
   public void WriteXmlSchema ( TextWriter writer);
   public void WriteXmlSchema ( XmlWriter writer );

   public void ReadXmlSchema  ( Stream stream );
   public void ReadXmlSchema  ( string fileName );
   public void ReadXmlSchema  ( TextWriter writer);
   public void ReadXmlSchema  ( XmlWriter writer );

   public void InferXmlSchema ( Stream stream, string[] namespaces );
   public void InferXmlSchema ( string fileName, string[] namespaces );
   public void InferXmlSchema ( TextWriter writer, string[] namespaces );
   public void InferXmlSchema ( XmlWriter writer, string[] namespaces );
}
```

Typisierte DataSets
Persistente Objekte

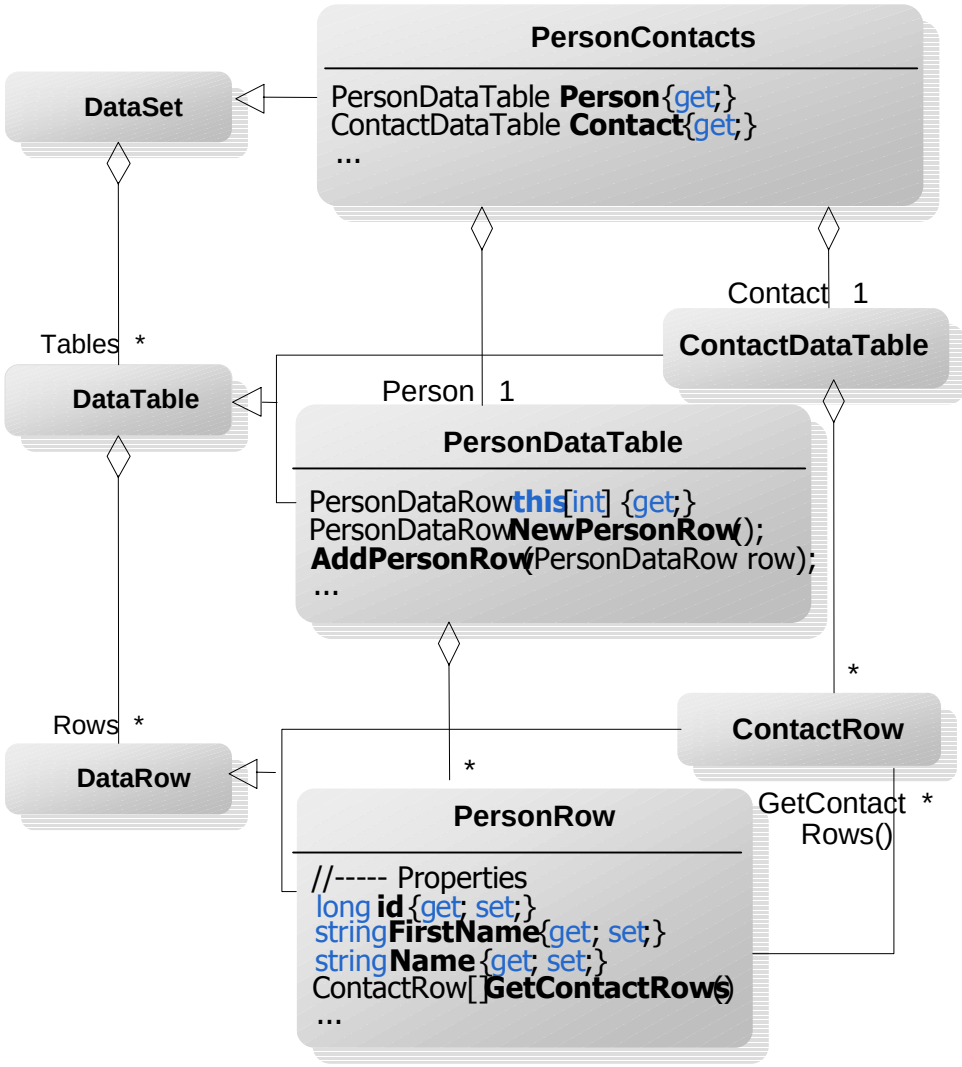
Typisierte DataSets

- *Typisierte DataSets* erlauben typisierten Datenzugriff
- Werkzeug xsd.exe generiert aus XML-Schema eigene Klassen

```
> xsd.exe personcontact.xsd
  /dataset
```

- Klassen definieren Properties für typisierten Zugriff auf Zeilen, Spalten und Relationen
- Statt Schemas zu entwickeln kann man sich dieses auch generieren lassen

```
xsd.exe <assembly>.dll|.exe
[/type: [...]]
```



Beispiel zu typisierten DataSets

School of Engineering

- Zugriffe mit herkömmlichem DataSet

```
DataSet ds = new DataSet("PersonContacts");
DataTable personTable = new DataTable("Person");
...
ds.Tables.Add(personTable);
DataRow person = personTable.NewRow();
personTable.Rows.Add(person);
person["Name"] = "Beer";
...
person.GetChildRows("PersonHasContacts")[0]["Name"] = "Beer";
```

- Zugriffe mit typisiertem DataSet

```
PersonContacts typedDS = new PersonContacts();
PersonTable personTable = typedDS.Person;
Person person = personTable.NewPersonRow();
personTable.AddPersonRow(person);
person.Name = "Beer";
...
person.GetContactRows()[0].Name = "Beer";
```

Persistente Objekte

- Vorteil
  - Man kann mit "normalen" Objekten innerhalb des Programms arbeiten
  - Überprüfung der Typen durch Compiler
  - Serialisierung wird einfacher (z.B. für Web Services)

- Nachteil:
  - müssen generiert werden

    http://blah.winsmarts.com/2006/06/02/are-you-a-business-object-person-or-a-dataset-person.aspx

- Ab .NET 4: MS Entity Framework
  - Version 1 stark kritisiert; Version 2 "Managed Objects" Konzept

- NHibernate (Portierung von Hibernate auf .NET)
  - http://blogs.hibernatingrhinos.com/nhibernate/archive/2008/04/01/your-first-nhibernate-based-application.aspx

Zusammenfassung

School of Engineering

- Verbindungsorientierter Datenzugriff
  - objektorientierte Schnittstelle abstrahiert von Datenquelle
  - Zugriff auf Datenbank über SQL-Kommandos
  - geeignet für
    - *wenig parallele Zugriffe und kurze Transaktionen*
    - *jederzeit aktuelle Daten*

- Verbindungsloser Datenzugriff
  - geeignet für viele parallele Zugriffe
  - DataSet ist Hauptspeicherdatenbank
  - DataAdapter für Verbindung zur Datenquelle
  - starke Integration mit XML
  - gut integriert in .NET-Klassenbibliothek (z.B.: WebForms, WinForms)

Fragen?

Anhang: Load- und StoreTable in DataSet

```
static string path = @"d:\My Documents\ZHW\DNET\DNET8\";
static string strConnection = @"Provider=Microsoft.Jet.OLEDB.4.0;
  Data Source=" + path+"Contacts.mdb; Persist Security Info=False;";

// ladet den Datensatz
static void LoadTable(DataSet ds, string tableName) {
  OleDbConnection con = new OleDbConnection (strConnection);
  //----- SelectCommand setzen
  OleDbDataAdapter adapter =
    new OleDbDataAdapter("SELECT * FROM " + tableName, con);
  //-----füge Schema Information automatisch hinzu
  adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;
  adapter.Fill(ds,tableName);
  if (ds.HasErrors) {ds.RejectChanges();
      throw new Exception("Error Loading Data");
  }
  else ds.AcceptChanges();
  adapter.Dispose();
  Console.WriteLine("Loaded Table:"+tableName);
}

static void StoreTable(DataSet ds, string tableName) {
  OleDbConnection con = new OleDbConnection (strConnection);
  //----- SelectCommand setzen, damit der OleDbCommandBuilder
  automatisch
  // Insert-, Update- und Delete-Kommandos generieren kann
  OleDbDataAdapter adapter =
    new OleDbDataAdapter("SELECT * FROM " + tableName, con);
  OleDbCommandBuilder cmdBuilder = new OleDbCommandBuilder(adapter);
  cmdBuilder.QuotePrefix = "[";cmdBuilder.QuoteSuffix = "]";
  //----- Daten speichern!
  adapter.Update(ds, tableName);
  adapter.Dispose();
  Console.WriteLine("Stored Table:"+tableName);
}
```

```
static void Print(DataSet ds) {
    Console.WriteLine("DataSet {0}:", ds.DataSetName);
    Console.WriteLine();
    foreach(DataTable t in ds.Tables) {
      Print(t);
      Console.WriteLine();
    }
  }

  static void Print(DataTable t) {
    //---- Tabellenkopf
    Console.WriteLine("Tabelle {0}:", t.TableName);
    foreach(DataColumn col in t.Columns) {
      Console.Write(col.ColumnName + "|");
    }
    Console.WriteLine();
    for (int i=0; i < 40; i++) {Console.Write("-");}
    Console.WriteLine();

    //---- Daten
    int nrOfCols = t.Columns.Count;
    foreach(DataRow row in t.Rows) {
      for(int i=0; i < nrOfCols; i++) {
        Console.Write(row[i]); Console.Write("|");
      }
      Console.WriteLine();
    }
  }
```