

.NET: ADO.NET

© J. Heinzelreiter
Version 5.3

früher war die Kopplung zu Datenbank sehr eng (Ausorkonzept)

ADO.NET – Designziele

- Trennung von **Datenzugriff** und **Datenmanipulation**.
 - Daten können in **Datenbehältern** (DataSets) gespeichert werden,<sup>In-Memory Darstellung
1 der Daten</sup>
 - Verbindung zur Datenbank muss **nur für die Dauer des Zugriffs** bestehen.
- Unterstützung **mehrschichtiger Anwendungen**.
 - Transport der Daten zwischen den **Schichten mit DataSets**.
 - Unterstützung von **Connection-Pooling**.
 - Unterstützung von **optimistischem Sperren**.<sup>traditionelles Speeren
im Web nicht
gut</sup>
- Umfassende **Unterstützung von XML**
- Datenaustausch zwischen **verschiedenen Plattformen** und **Architekturen** (Web Services).

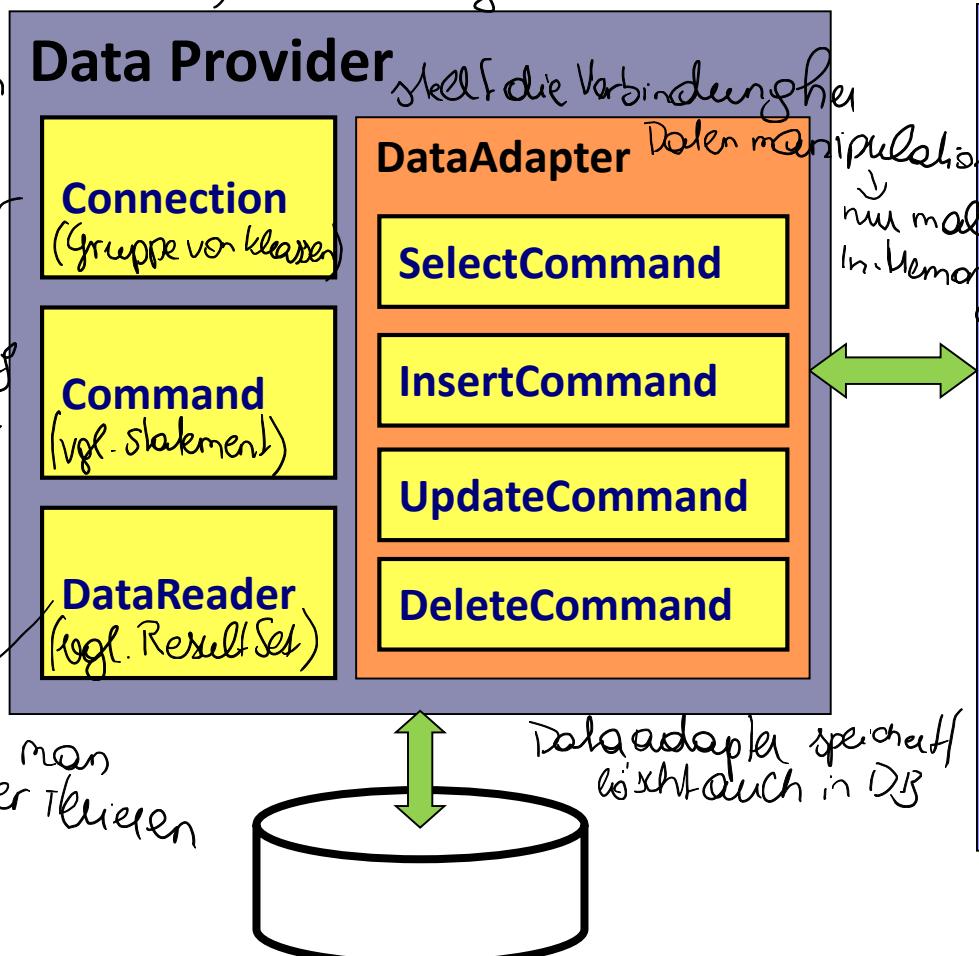
Daten müssen serialisiert sein

Architektur

sowie bei JDBC werden
IF verwendet
und der Driver
hält die
Implementierung

SQL
Abfrage

Bonn man
drüber klauen



DataSet nur Möglichkeit der
Repräsentation

DataSet

DataTableCollection

DataTable

DataRowCollection

DataColumnCollection

ConstraintCollection

DataRelationCollection

XML

- gut zu
verwenden
in UI

Veränderungen
am DataSet
nicht direkt
in DB

Data-Provider: Konzept

■ Geschichte

■ ODBC:

- Gemeinsame **funktionsorientierte C-API**.
- **ODBC-Treiber** abstrahiert Schnittstelle zu konkrem DBMS.

Ab der DB soll bei der Implementierung keine Rolle spielen

■ ADO

- Zugriff erfolgt über **definierte COM-Schnittstellen**.
- **OLE-DB-Provider** abstrahiert Schnittstelle zu DBMS.

ADO-Ziel: Sprachunabhängige DB-Schnittschicht

■ ADO.NET

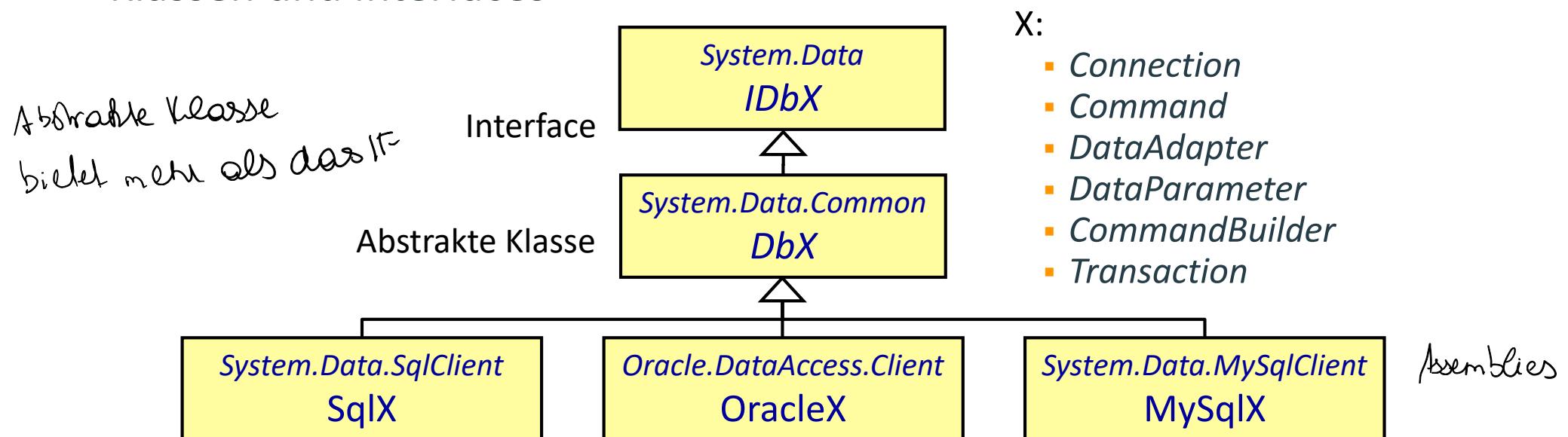
„für versch. DBs“

- DBMS-spezifischer **Data-Provider** direkt verwendet werden.
- Optimierungen für konkretes DBMS sind möglich.
- Datenbank-Unabhängigkeit geht aber teilweise verloren.
- Durch Verwendung von Interfaces und abstrakten Basisklassen kann Datenbankspezifischer Teil sehr klein gehalten werden.

Ado.net erlaubt direkten Zugriff auf die Klassen des Data Providers

Data-Provider: Implementierungen

- Implementierungen von Data-Providern
 - *SqlServer*: in Full Framework vorhanden
 - *Oracle*: ODB.NET
 - *MySQL*: Connector/.NET
 - *DB2*: DB2.NET
 - *ODBC/OleDb*
 - Wrapper um bestehende Treiber
→ nativer Code
- Klassen und Interfaces



Connection – Datenbankverbindungen

- Repräsentiert eine Verbindung zu einer DB.
 - ConnectionString: Parameter für Verbindungsaufbau.

- Server=myhost; User ID=sa; Password=susan;
Pooling=true; Max Pool Size=50;
 - Source=(LocalDB)\MSSQLLocalDB; AttachDbFilename=C:\Db\MyDb.mdf;
 - DataSource=tcp:myserver.database.windows.net,1433;
Initial Catalog=FhQuotesDb; ID=sa; Password=susan;

ConnectionString verschlüsseln

- Open() und Close():

```
using ( IDbConnection conn = new SqlConnection(myConnStr)) {  
    conn.Open();  
    using ( IDbCommand cmd = new SqlCommand(sqlQuery, conn)) {  
        ...  
    }  
} // conn.Dispose() → conn.Close()
```

- ADO.NET unterstützt standardmäßig **Connection Pooling**.

wenn mit Laptop im UE-Kurs nochmal schaun ob im Visual Studio alles mit installiert ist

ConnectionString verschlüsseln

physische conn bleibt offen aber die logische geschlossen und zurück in den Pool gegeben

DataReader – Iteration durch Datensätze

ExecuteReader

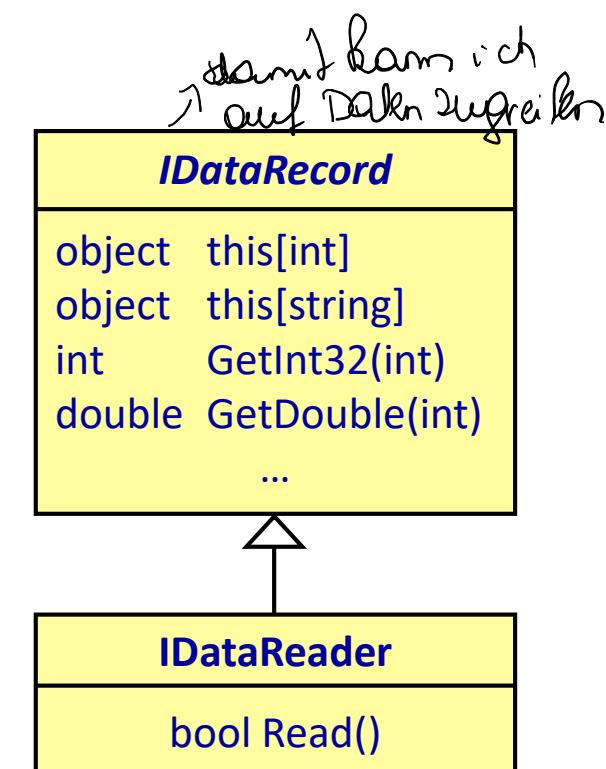
- Liefert ein Objekt, das die Interfaces **IDataReader** und **IDataRecord** implementiert.
- Mit **IDataReader** kann durch **Ergebnis** iteriert werden.
- IDataRecord** ermöglicht den Zugriff auf Attributwerte.

```
string sql = "SELECT name, age FROM Person";
IDbCommand selectCmd = new SqlCommand(sql, conn);
IDataReader reader = selectCmd.ExecuteReader();

while (reader.Read()) {
    string name = (string)reader[0];
    int    age   = (int)reader["age"];
}
```

... aber Objekt darum der Quot

sind auch **IDisposable**, hauptsächlich mit **using** arbeiten
(**DataReader**, **Command**)



Command – Datenbank-Abfragen

■ ExecuteNonQuery:

- Schreibende DB-Kommandos (*insert, update, delete*).
- Rückgabewert ist die Anzahl der betroffenen Datensätze.

```
string sql = "UPDATE Person SET name='Franz'";
IDbCommand updCmd = new SqlCommand(sql, conn);
int rowsAffected = updCmd.ExecuteNonQuery();
```

■ ExecuteScalar:

- Kommandos mit skalarem Rückgabewert.
- Rückgabewert muss in passenden Typ konvertiert werden.

```
string sql = "SELECT COUNT(*) FROM Person"; max, avg, min, ...
IDbCommand countCmd = new SqlCommand(sql, conn);
int noOfPersons = (int)countCmd.ExecuteScalar();
```

Abfragen mit Parametern

um SQL Injektion zu vermeiden

- Abfragen können mit Parametern versehen werden.

```
string sql = "UPDATE Person SET age=age+1 WHERE name = ?";  
IDbCommand updCmd = new OleDbCommand(sql, conn);
```

- SQLServer und Oracle unterstützen benannte Parameter. *SQL und MySQL Syntax*

```
string sql = "UPDATE Person SET age=age+1 WHERE name=@name";  
IDbCommand updCmd = new SqlCommand(sql, con);
```

- Eigenschaften von Parametern müssen definiert werden.

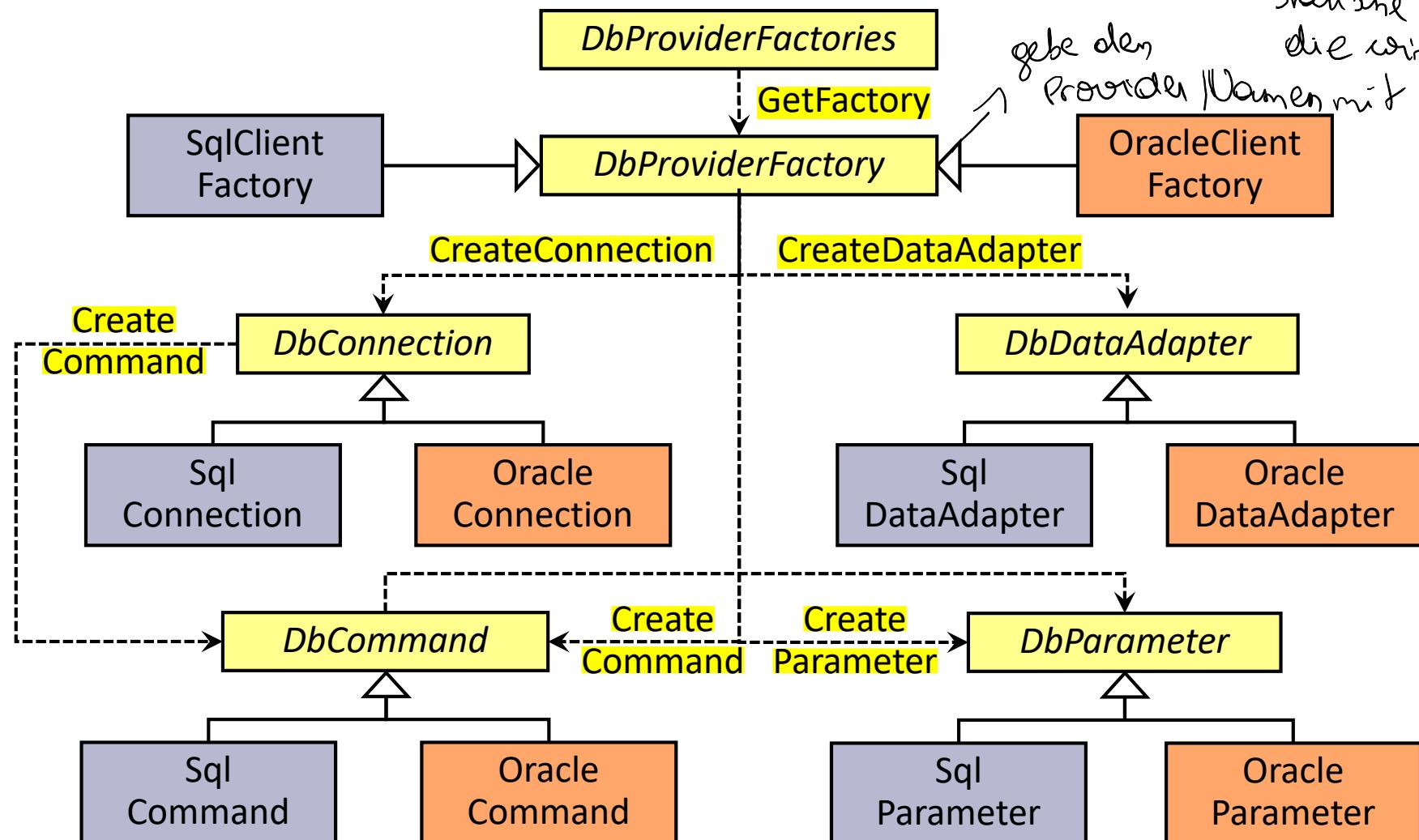
```
SqlParameter nameParam = new SqlParameter("@name", SqlDbType.VarChar);  
updCmd.Parameters.Add(nameParam);
```

- Parametern müssen vor Ausführung Werte zugewiesen werden.

```
nameParam.Value = "Mayr";  
updCmd.ExecuteNonQuery();
```

*updateParameters["name"]
geht auch*

Provider-unabhängige Programmierung (1)



gelbe sind
statische Klassen,
die wir verwenden
gebe den
Provider Namen mit

Provider-unabhängige Programmierung (2)

- Provider-abhängige Parameter in Konfigurations-Datei definieren

```
<configuration>
  <connectionStrings>
    <add name="MyDbConnection"
      connectionString="..." Server username , pWd
      providerName="System.Data.SqlClient"/>Top Level Namespace
  </connectionStrings>
</configuration>
```

- Verwendung im Code

- Factory aus Konfigurations-Parameter erzeugen
- Factory erzeugt Provider-abhängige Objekte
- Im Code werden Provider-unabhängige Interfaces verwendet

```
var connSettings = ConfigurationManager.ConnectionStrings["MyDbConnection"];
DbProviderFactory dbfactory =
  DbProviderFactories.GetFactory(connSettings.ProviderName);
IDbConnection dbconn = dbfactory.CreateConnection();
dbconn.ConnectionString = connSettings.ConnectionString;
dbconn.Open();
IDbCommand dbcomm = dbconn.CreateCommand();
```

Asynchrone Programmierung

- Für zeitaufwändige Datenbank-Operationen bietet ADO.NET asynchrone Methoden an.
- Beispiel:

```
using (DbConnection conn = dbFactory.CreateConnection()) {  
    await conn.OpenAsync();  
    using (DbCommand selCmd = conn.CreateCommand()) {  
        selCmd.CommandText = "SELECT name, age FROM Person";  
        selCmd.Connection = conn;  
        using (DbDataReader reader = await selCmd.ExecuteReaderAsync()) {  
            while (await reader.ReadAsync())  
                Process(new Person((string)reader["name"],  
                                  (int)reader["age"]));  
        }  
    }  
}
```

Asynchrone Methoden
nur in den abstrakten Klassen (DbX), nicht in den Interfaces (IDbX)

Transaktionen

- Transaktionen sind unteilbare Aktionen in der DB.
 - **Commit:** alle DB-Aktionen werden gemeinsam durchgeführt.
 - **Rollback:** alle DB-Aktionen werden rückgängig gemacht.
- Beispiel:

```
DbCommand cmd = new SqlCommand(sql, connection);
DbTransaction trans =
    connection.BeginTransaction(IsolationLevel.ReadCommitted);
cmd.Transaction = trans;

try {
    cmd.CommandText = "UPDATE ..."; cmd.ExecuteNonQuery();
    cmd.CommandText = "UPDATE ..."; cmd.ExecuteNonQuery();
    trans.Commit();
}
catch(Exception e) {
    trans.Rollback();
}
```

Ambiente Transaktionen

- .NET 2.0 definiert eine **Transaktions-API**, mit der *ambiente Transaktionen* definiert werden können (*System.Transactions*)

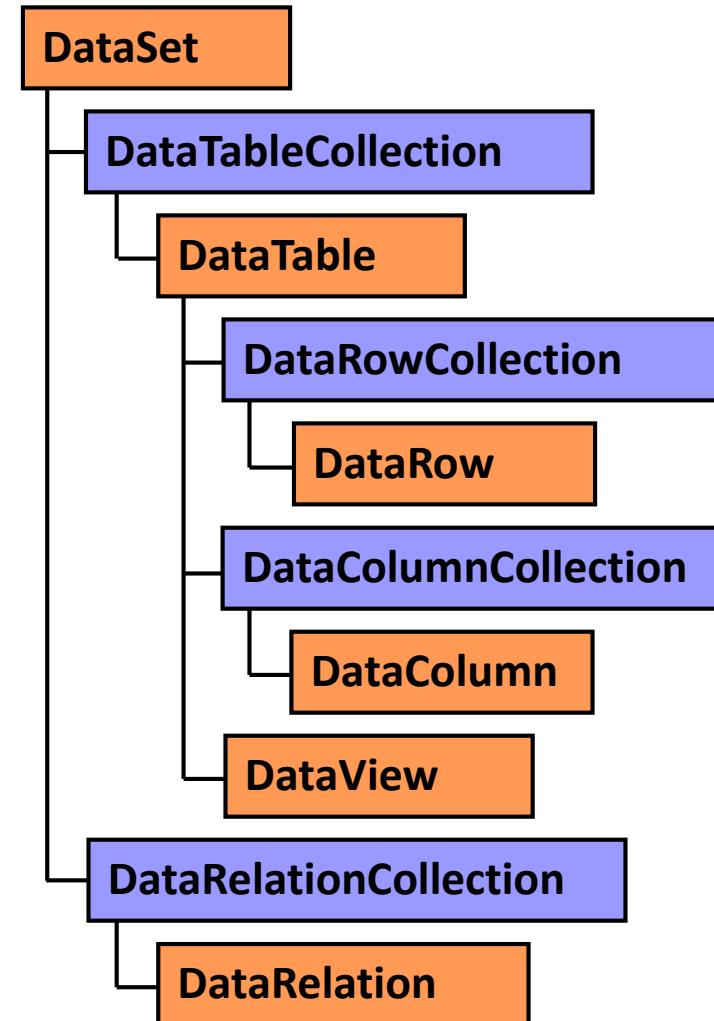
```
DbCommand cmd = new SqlCommand(sql, connection);
using (TransactionScope txScope = new TransactionScope()) {
    cmd.CommandText = "UPDATE ..."; cmd.ExecuteNonQuery();
    cmd.CommandText = "UPDATE ..."; cmd.ExecuteNonQuery();
    txScope.Complete();    → werden alle verständigt um es zu machen
}
```

- Alle Anweisungen zwischen dem Aufruf des Konstruktors und der Methode Dispose() werden zu einer Transaktion zusammengefasst.
- In Dispose() wird die Transaktion bestätigt, falls vorher Complete() aufgerufen wurde; sonst wird sie zurückgenommen.
- Vorteil: Ressourcen, die diese API unterstützen, beteiligen sich automatisch an der ambienten Transaktion.
- Ressource muss „Auto-Enlistment“ unterstützen (z.B. SqlServer, Oracle, ...)

DataSets

- DataSets bestehen aus *mehreren Tabellen*.
- Zwischen Tabellen können *Beziehungen* definiert werden.
- DataSets sind unabhängig von der *Datenherkunft*.
- *Data Adapter* stellen Verbindung zur Datenquelle her.
- DataSets sind ein *Offline-Container* für Daten.
- DataSets können *offline modifiziert* werden.

NorthWindDataSet.cs



Untyped Datasets

- Erzeugung eines *Untyped* Datasets:

```
DataSet ds = new DataSet("Shop");
ds.Tables.Add("Article");
ds.Tables["Article"].Columns.Add("ID",      typeof(int));
ds.Tables["Article"].Columns.Add("Price",   typeof(double));
...
DataColumn[] keys = new DataColumn[1];
keys[0] = ds.Tables["Article"].Columns["ID"];
ds.Tables["Article"].PrimaryKey = keys;
```

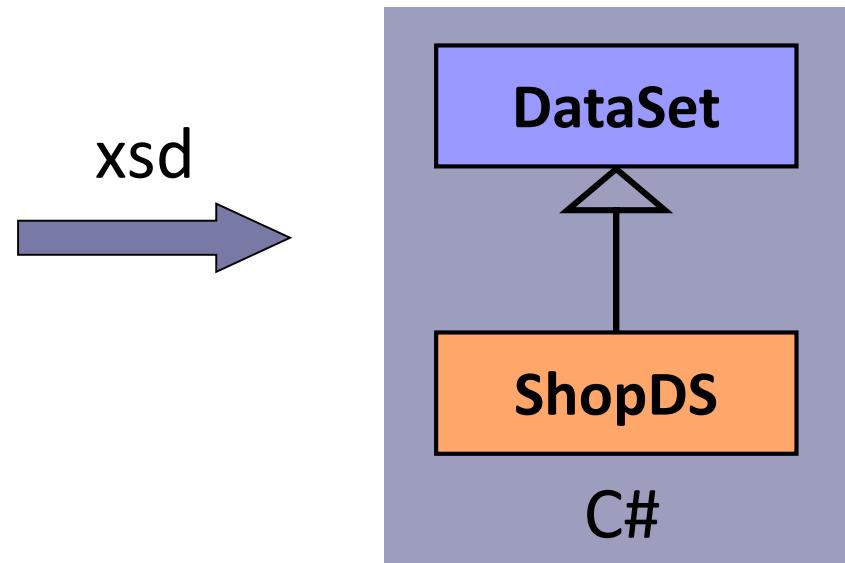
- Verwendung eines Untyped Datasets:

```
DataTable article = ds.Tables["Article"];
foreach(DataRow r in article.Rows)
    Console.WriteLine("ID={0}, price={1}",
                      r["ID"], r["Price"]);
```

Typed Datasets

- Erzeugung eines *Typed Datasets*:

```
<?xml version="1.0"?>
<xsd:element name="Shop">
  <xsd:element name="Article">
    <xsd:element name="id"
      type="xsd:int"/>
  </xsd:element>
</xsd:element>
```



- Verwendung eines Typed Datasets

```
ShopDS.ArticleDataTable article = shopDS.Article;
foreach (ShopDS.ArticleRow row in article.Rows)
  Console.WriteLine("ID={0}, Price={1}",
    row.ID, row.Price);
```

Erstellen von Typed DataSets mit VS .NET

The screenshot shows the XML Schema Editor in Visual Studio .NET. On the left, the **Komponenten-palette** (Component palette) is open, displaying various data-related components like Pointer, DataSet, OleDbDataAdapter, etc. In the center, the **XML Schema Editor** displays three typed datasets:

- Artikel** (Artikel):
 - E Artikelname string
 - E Artikel-Nr int
 - E Auslaufartike boolean
 - E BestellteEinh short
 - E Einzelpreis decimal
- Bestellungen** (Bestellungen):
 - E Bestelldatum dateTime
 - E Bestell-Nr int
 - E Bestimmung string
 - E Empfänger string
 - E Frachtkosten decimal
- Bestelldetails** (Bestelldetails):
 - E Anzahl short
 - E Artikel-Nr int
 - E Bestell-Nr int
 - E Einzelpreis decimal
 - E Rabatt float

Dashed lines connect the three datasets, indicating they share a common schema.



Manipulation von Datasets

- Einfügen einer neuen Zeile

```
ShopDS.ArticleRow r = shopDS.Article.NewArticleRow();
r.ID      = 77;
r.Price   = 55.55;
shopDS.Article.AddArticleRow(r);
```

- Aktualisieren einer Zeile

```
ShopDS.ArticleRow r = shopDS.Article.FindBy_Article_ID(88);
r.BeginEdit();    // optional: Events werden erst bei EndEdit() gefeuert.
r.Price = 99.99;
r.EndEdit();
```

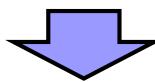
- Löschen einer Zeile

```
ShopDS.ArticleRow r =
    shopDS.Article.FindBy_Article_ID(99);
r.Delete();
```

Versionsinformation in DataSets

- In DataSets werden drei Versionen der Datensätze gespeichert:
 - *Original*: Ursprüngliche Werte, nach AcceptChanges() werden aktuelle Werte in ursprüngliche Werte kopiert.
 - *Proposed*: Änderungen zwischen BeginEdit() und EndEdit().
 - *Current*: Änderungen werden in aktueller Version gespeichert.
- Beispiel:

```
foreach (ShopDS.ArticleRow r in a.article.Rows)
    int id = r.RowState == DataRowState.Deleted ?
        (int)r[article.IDColumn, DataRowVersion.Original] : r.ID;
    Console.WriteLine("{0} {1}", id, r.RowState);
```



66 Unchanged
77 Added
88 Modified
99 Deleted

Manipulation von DataSets

- Übernahme/Rücknahme der Änderungen

```
try {  
    ...  
    shopDS.AcceptChanges();  
} catch (Exception) {  
    shopDS.RejectChanges();  
}
```

- *AcceptChanges: Original = Current*
- *RejectChanges: Current = Original*
- Verbindung zu Adaptern
 - Bei Datenübernahme werden Daten in Originalversion übernommen und *Current = Original* gesetzt:

```
articleAdapter.Fill(shopDS.Article);
```

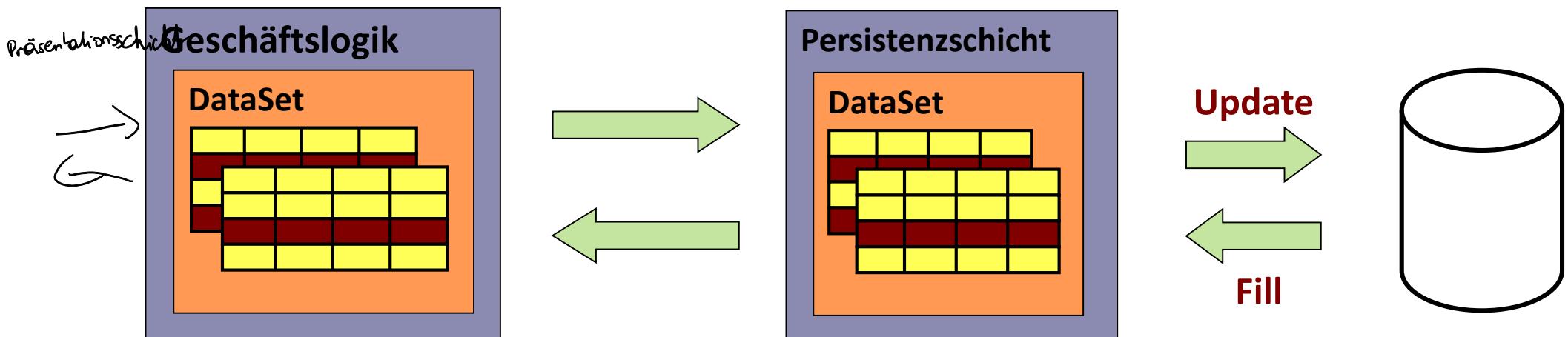
- Beim Zurückschreiben in Datenbank wird *AcceptChanges* aufgerufen.

```
articleAdapter.Update(shopDS.Article)
```

Austausch von Datasets

womit empfehlenswert für die Projektarbeit
bricht aus .NET raus, sehr schlecht

- Datasets sind für den Austausch von Daten zwischen **verschiedenen Schichten** ausgelegt.
- Datasets können einfach **in XML konvertiert werden**.
- Einfache Möglichkeit zum **Datenaustausch mit Web-Services**.
- Wesentliche Einschränkung: Datenaustausch mit Datasets ist nur innerhalb der **.NET-Plattform praktikabel**.



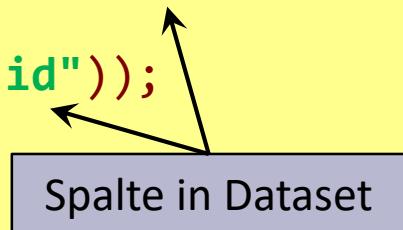
Daten-Adapter – Verbindung zur Datenquelle

- **Befüllen** von Datasets mit Daten

```
SqlDataAdapter articleAdapter = new SqlDataAdapter();
articleAdapter.SelectCommand =
    new SqlCommand("SELECT * FROM Article", connection);
articleAdapter.Fill(shopDS.Article);
```

- **Synchronisation** mit der Datenquelle

```
SqlDataAdapter articleAdapter = new SqlDataAdapter();
SqlCommand updCmd = new SqlCommand(
    "UPDATE Article SET Price=@price WHERE ID=@id", connection);
updCmd.Parameters.Add(
    new SqlParameter("@price", SqlDbType.Double, 0, "price"));
updCmd.Parameters.Add(
    new SqlParameter("@id", SqlDbType.Integer, 0, "id"));
articleAdapter.UpdateCommand = updCmd;
articleAdapter.Update(shopDS.Article);
```



Automatische Generierung von Abfragen

- Mit einem Daten-Adapter kann ein **CommandBuilder** verbunden werden.

```
adapter = new SqlDataAdapter(  
    "SELECT id, price, FROM Article", conn);  
commandBuilder = new SqlCommandBuilder(adapter);
```

- Der **CommandBuilder** erzeugt die **Update-, Insert- und Delete-Abfragen**, die zur Aktualisierung eines Datasets benötigt werden.

```
Console.WriteLine(commandBuilder.GetInsertCommand()  
    .CommandText);
```



```
INSERT INTO Article (id, price) VALUES (?, ?));
```

Daten-Adapter: Update-Verhalten

- ADO.NET verwendet *optimistisches Locking*.

- Datensätze werden beim/nach Befüllen des DataSets nicht gelockt.
- Nur während des Synchronisierens mit der Datenbank wird gelockt.

- *CommandBuilder* erzeugt Abfragen, die nur (von anderen Benutzern) unveränderte Datensätze überschreiben.

- Sonst wird eine *DbConcurrencyException* geworfen.

```
updCmd.CommandText = "UPDATE Article SET id = @id, price = @price"
                  + " WHERE (id = @origId) AND (price = @origPrice)";
updCmd.Parameters.Add(new SqlParameter("@id", SqlDbType.Integer, 0,
                                      "id"));
updCmd.Parameters.Add(new SqlParameter("@price", SqlDbType.Currency,
                                      0, "price"));
updCmd.Parameters.Add(new SqlParameter("@origId",
                                      SqlDbType.Integer, ... "id", DataRowVersion.Original, ...));
updCmd.Parameters.Add(new SqlParameter("@origPrice", SqlDbType.Currency,
..., "price", DataRowVersion.Original, ...));
```

optimistic locking: . beim Lesen eines Datensatz wird dieser gelockt und bei Speichern wieder freigabe

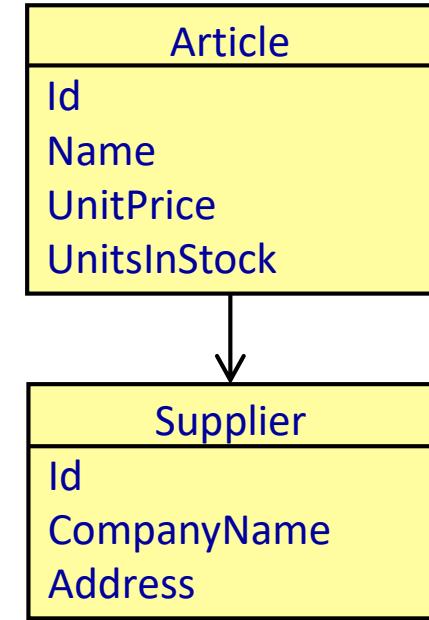
in Web ein Problem. da zu viel Locking

Optimistic: erst beim Synchronisieren wird gefragt ob sich was ändert

was steht vorher und was ist in DB? hat sich was geändert? so wird es erkannt

Domänenklassen

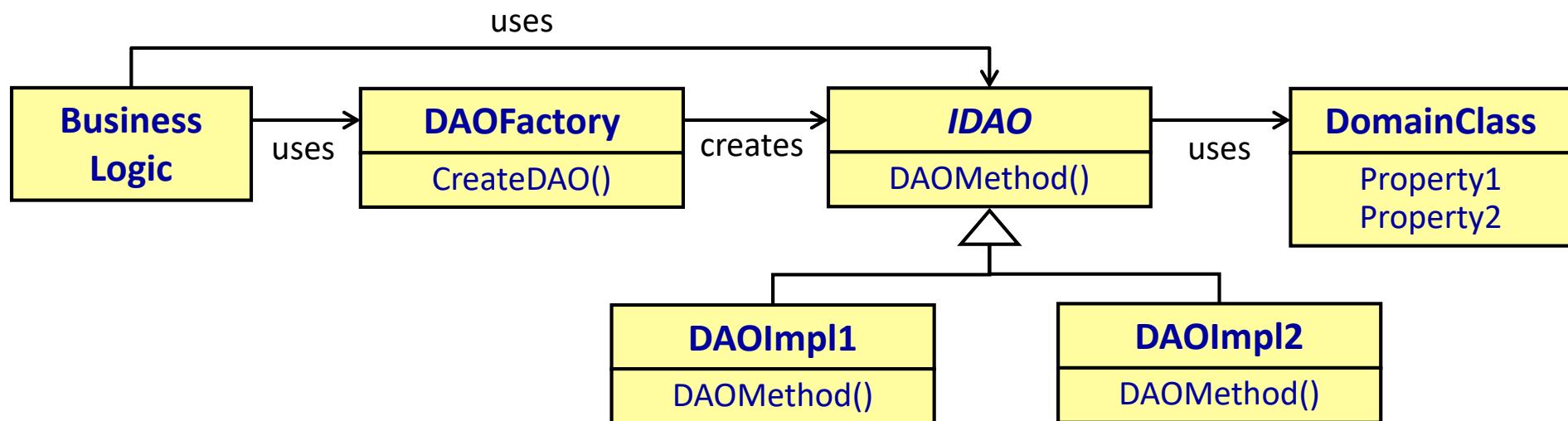
- Zur Repräsentation der Daten im Hauptspeicher können
 - Datasets („change sets“) oder
 - Domänenklassen verwendet werden.
- Domänenklassen sind einfache Klassen (POCOs – Plain Old CLR Objects)
 - mit Konstruktoren und Propertys
 - und optionalen Referenzen zu anderen Domänenklassen.
- Domänenklassen sind völlig technologieunabhängig.
- Die Manipulation der Daten wird in eigenen Datenzugriffsklassen (DAOs) durchgeführt.



Das DAO-Muster

- Problem: DB-Zugriffsklassen sind technologieabhängig.
- Abhilfe:
 - Datenzugriffsobjekt implementiert ein Interface.
 - Factory liefert das gewünschte Datenzugriffsobjekt.
 - Nur in Datenzugriffsobjekten werden ADO.NET-Klassen verwendet
 - Andere Schichten verwenden ausschließlich DAO-Interfaces.

Datenzugriffsoperationen
müssen selbst überlegt
werden

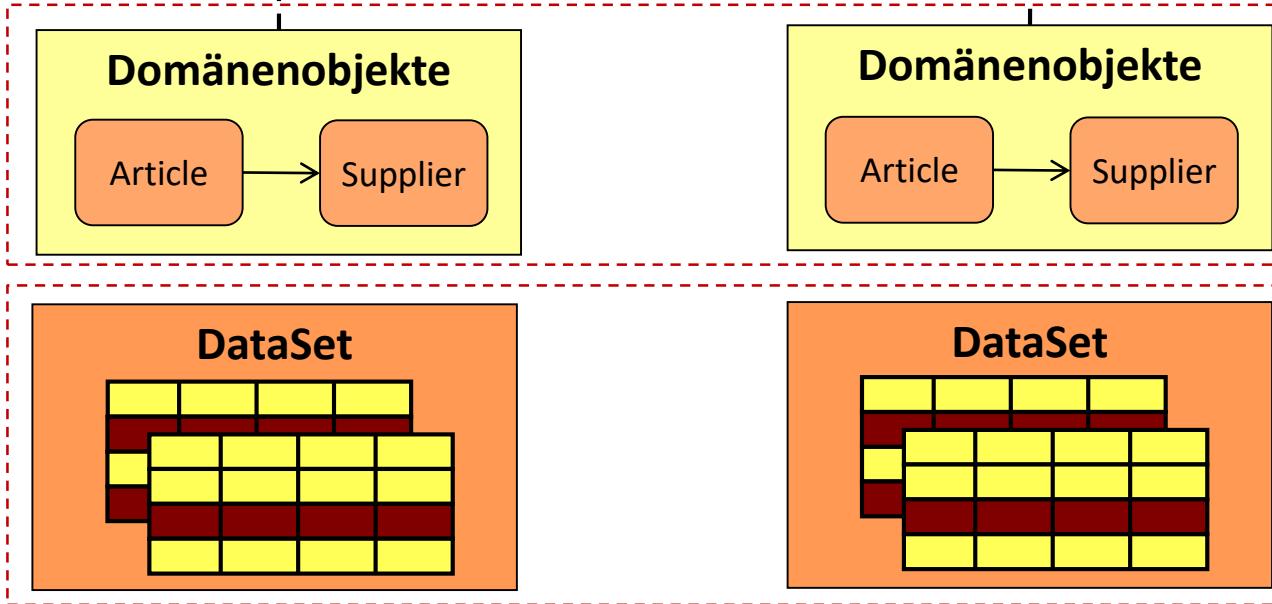


Domänenklassen/Datasets: Rolle in der Architektur

- Domänenklassen und Datasets sind einfach serialisierbar.
- Sie werden zum Transport der Daten zwischen den Schichten eingesetzt.



- Variante 1:
- Variante 2:



Pocos können
einfach in JSON
serienisiert werden