FH OBERÖSTERREICH

.NET: ADO.NET

Version 5.3

ADO.NET – Designziele

Unterschied Web und Desktop Client: Web mehr Clients und größere Distanz

UPDATE, INSERT, DELETE

an der Oberfläche (Steuerelement)

- Trennung von Datenzugriff und Datenmanipulation.
  - Daten können in Datenbehältern (DataSets) gespeichert werden.
  - Verbindung zur Datenbank muss nur für die Dauer des Zugriffs bestehen.
- Unterstützung mehrschichtiger Anwendungen.

Präsentationsschicht getrennt von Zugangsschicht

  - Transport der Daten zwischen den Schichten mit DataSets.

auch in verschiedenen Prozessen (Serialisierung, ...)

  - Unterstützung von *Connection-Pooling*.

Teilen von Datenbankverbindungen

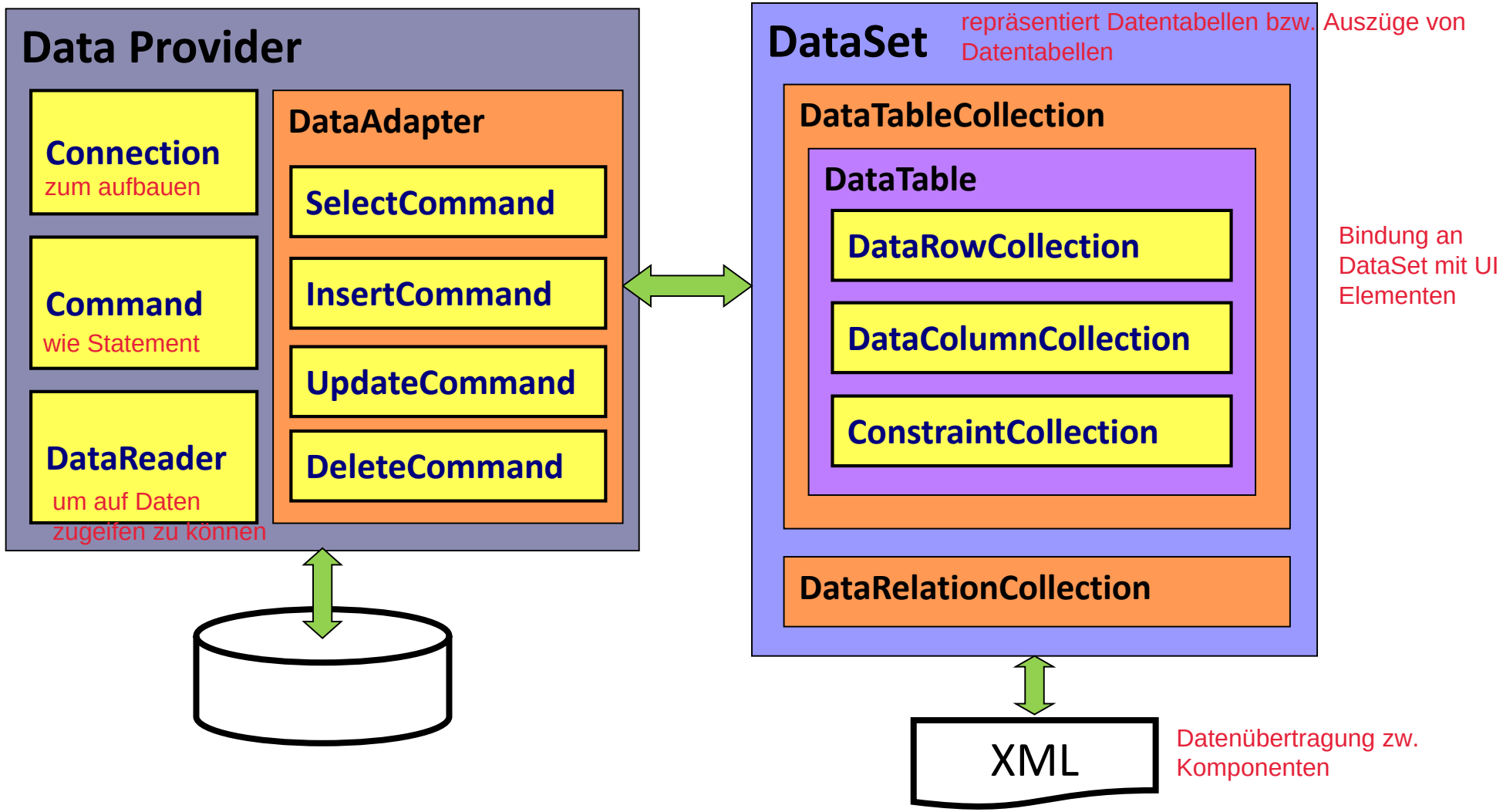
  - Unterstützung von optimistischem Sperren.
- Umfassende Unterstützung von XML
  - Datenaustausch zwischen verschiedenen Plattformen und Architekturen (Web Services).

in JDBC - ResultSet, Connection, Statement alles Interfaces im JDBC Treiber implementiert

für Angular Client 'eher' nicht

Architektur

nur Data Provider hat Zugriff auf Datenbank



repräsentiert Datentabellen bzw. Auszüge von Datentabellen

Bindung an DataSet mit UI Elementen

Datenübertragung zw. Komponenten

Data-Provider: Konzept

- Geschichte

  JDBC objektorientiert

  - ODBC: erster standardisierter DB Zugriffsmechanismus
    - Gemeinsame funktionsorientierte C-API.
    - ODBC-Treiber abstrahiert Schnittstelle zu konkretem DBMS.
  - ADO
    - Zugriff erfolgt über definierte COM-Schnittstellen.
    - OLE-DB-Provider abstrahiert Schnittstelle zu DBMS.

  ADO, ADO.Net - Konzepte unterschiedlich

- ADO.NET

  für versch. Datenbanken (MySQL, Oracle, ...)

  in JDBC kein Zugang zu Klassen der Treiber providerspezifischer Datenbanken

  - 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 Datenbank-spezifischer Teil sehr klein gehalten werden.

Data-Provider: Implementierungen

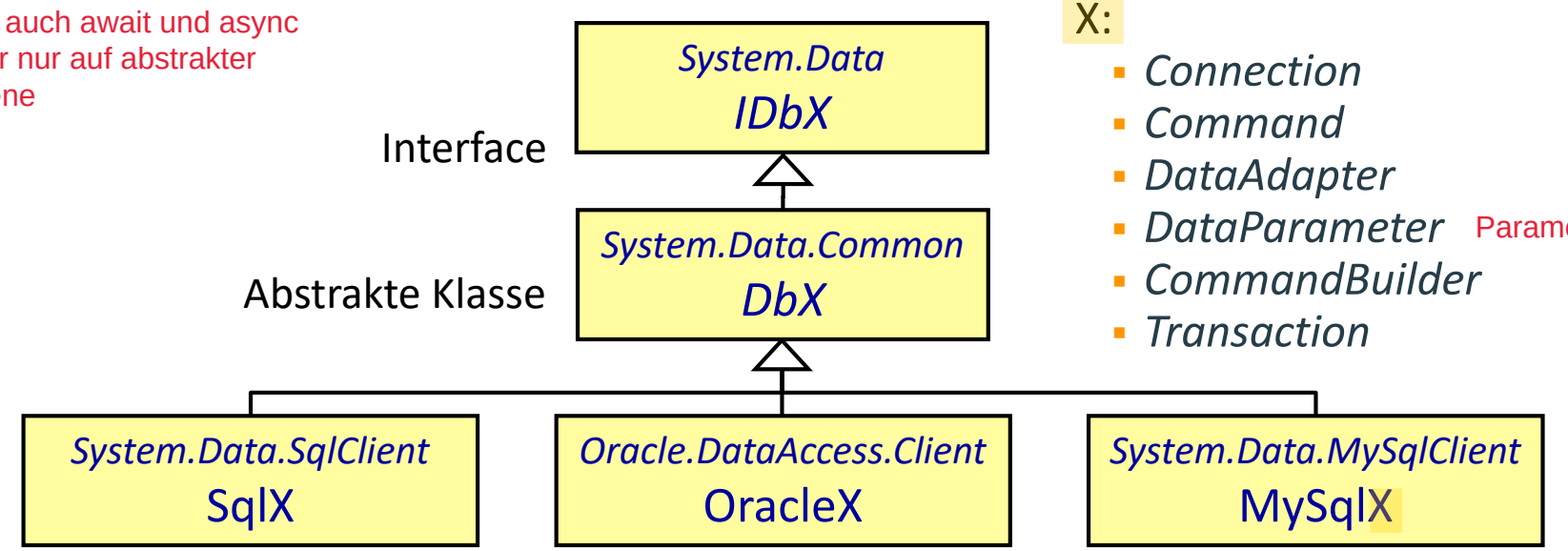
Gegenstück zu JDBC Treibern

- Implementierungen von Data-Providern (nur relational)
  - *SqlServer*
  - *Oracle:* ODB.NET
  - *MySQL:* Connector/NET
  - *DB2:* DB2.NET
  - *ODBC/OleDb*
    - Wrapper um bestehende Treiber
      → nativer Code

NoSQL über spezialisierte API's

- Klassen und Interfaces

gibt auch await und async aber nur auf abstrakter Ebene



X:

- *Connection*
- *Command*
- *DataAdapter*
- *DataParameter* (Parametrisierte Queryys)
- *CommandBuilder*
- *Transaction*

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;`
  - `Open()` und `Close()`:

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

Dispose so implementiert das es Close aufruft

SqlConnection implementiert IDisposable

  - ADO.NET unterstützt standardmäßig *Connection Pooling*.

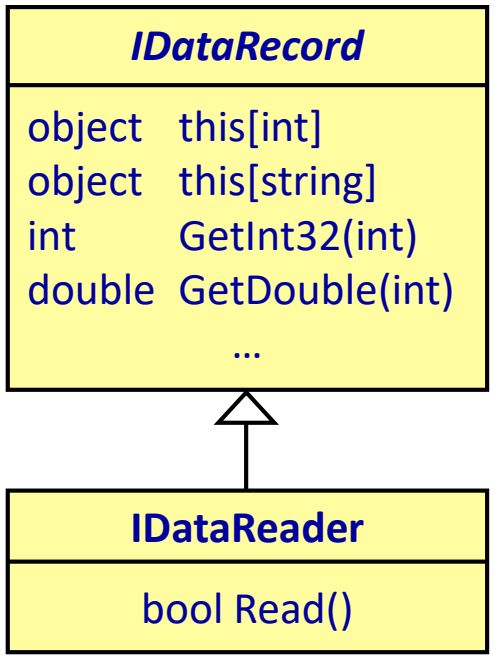
logisch geschlossen
nicht physisch geschlossen (kommt wieder in DataPool)

DataReader – Iteration durch Datensätze

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

```
string sql = "SELECT name, age FROM Person";
IDbCommand selectCmd = new SqlCommand(sql, conn);
   sind auch idisposable (using nutzen)
IDataReader reader = selectCmd.ExecuteReader();
                                  SQL Statement ausführen
while (reader.Read()) {  forward only iterator
  string name = (string)reader[0];
  int    age  = (int)reader["age"];
}
```

* und indexer [0] eher nicht verwenden



Command – Datenbank-Abfragen

- `ExecuteNonQuery`:
  - Schreibende DB-Kommandos (*insert*, *update*, *delete*).
  - Rückgabewert ist die Anzahl der betroffen 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";
IDbCommand countCmd = new SqlCommand(sql, conn);
int noOfPersons = (int)countCmd.ExecuteScalar();
```

ExecuteScalar - eine Zeile eine Spalte - darauf kann ich zugreifen

Abfragen mit Parametern

- 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.

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

- Eigenschaften von Parametern müssen definiert werden.

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

- Parametern müssen vor Ausführung Werte zugewiesen werden.

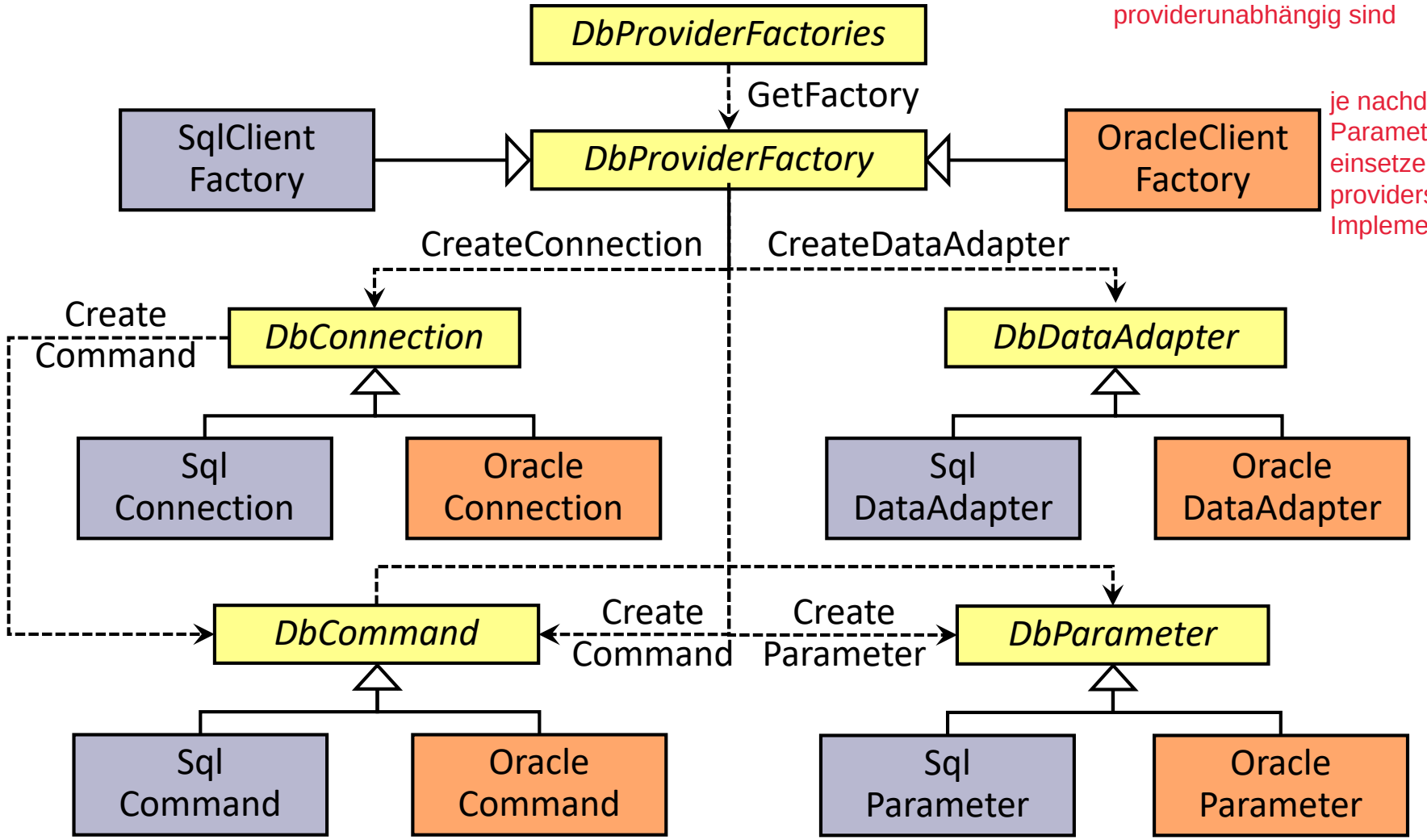
```
nameParam.Value = "Mayr";
updCmd.ExecuteNonQuery();
```

oder auch DbParameter db = command.CreateCommand();
db.ParameterName = p.Name;
db.Value = p.Value;
command.Parameters.Add(db);

Provider-unabhängige Programmierung (1)

Factory Method

gelb - abstrakte Klassen die providerunabhängig sind



je nachdem was ich beim Parameter als Provider einsetze bekomme ich providerspezifische Implementierung

Provider-unabhängige Programmierung (2)

- Provider-abhängige Parameter in Konfigurations-Datei definieren

```
<configuration>
  <connectionStrings>
    <add name="MyDbConnection"
         connectionString="…"
         providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

top level namespace

- 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 sind in den abstrakten Klassen (DbX) definiert, 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

auch mit transienten Transaktionen möglich

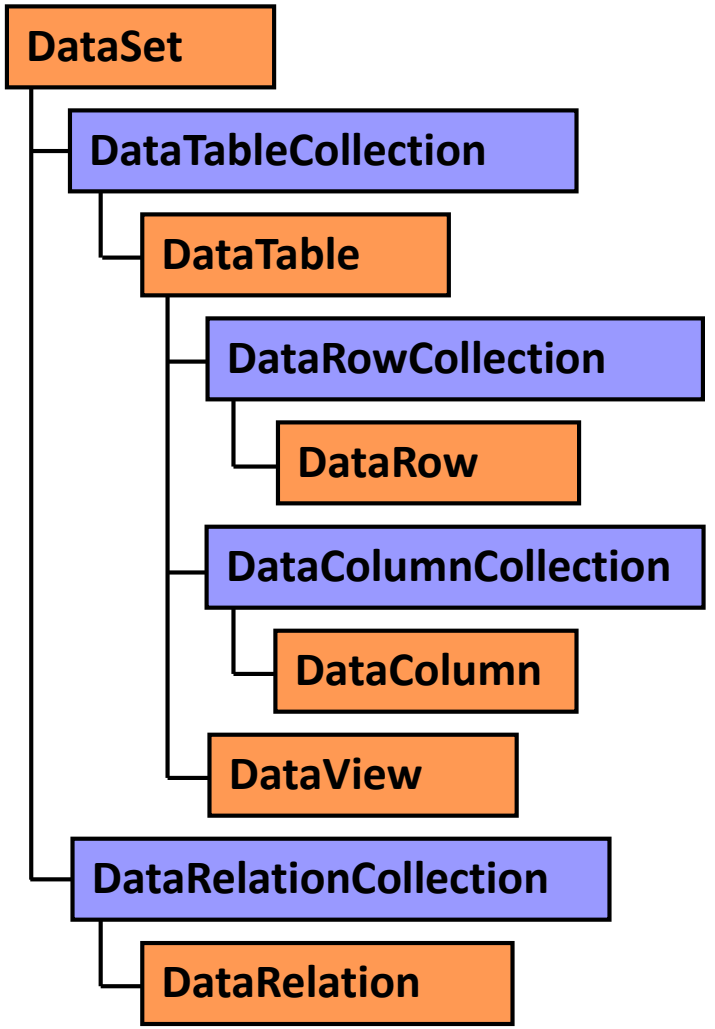
- .NET 2.0 definiert eine Transaktions-API, mit der *ambiente Transaktionen* definiert werden können *(System.Transactions)*

```
DbCommand cmd = new SqlCommand(sql, connection);
using (TransactionScope txScope = new TransactionScope()) {
  cmd.CommandText = "UPDATE …"; cmd.ExecuteNonQuery();
  cmd.CommandText = "UPDATE …"; cmd.ExecuteNonQuery();
  txScope.Complete();
}
```

  - 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“ unterstützten (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.



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;
```

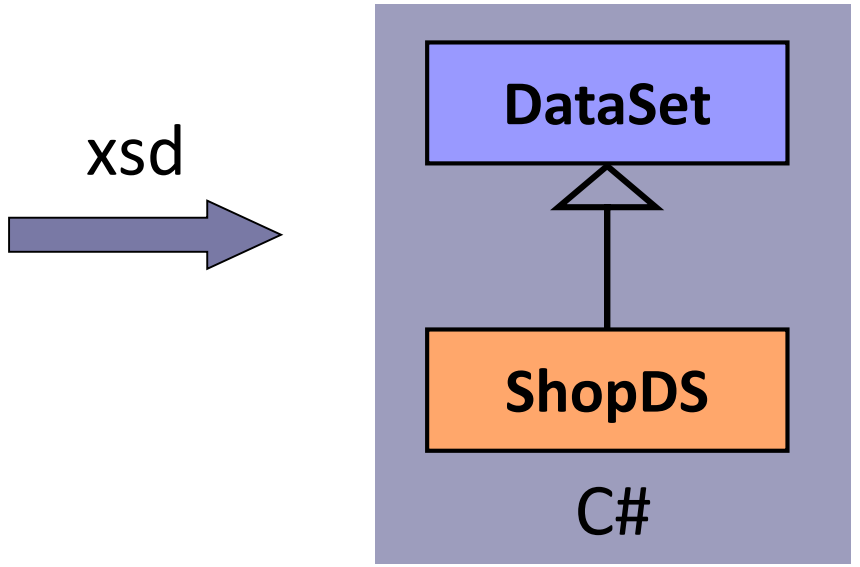
- Verwendung eines Untyped Datasets:

```
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:

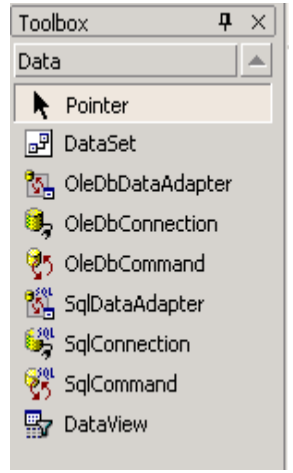
```
<?xml version="1.0"?>
<xsd:element name="Shop">
  ...
  <xsd:element name="Article">
    ...
    <xsd:element name="id"
                 type="xsd:int"/>
    ...
   </xsd:element>
</xsd:element>
```



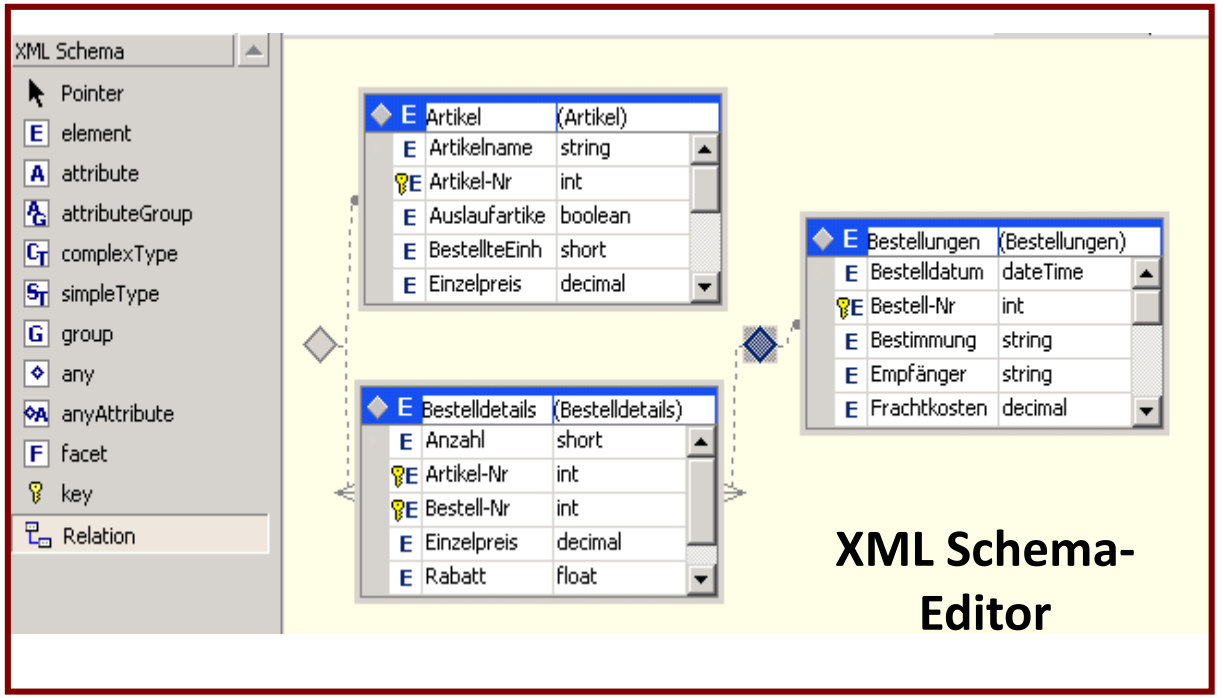
- Verwendung eines Typed Datasets

```
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



Komponenten-palette



XML Schema-Editor

instanzierte Komponenen

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();
}
```

  - *AccecptChanges: Original = Current*
  - *RejectChanges: Current = Original*

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

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

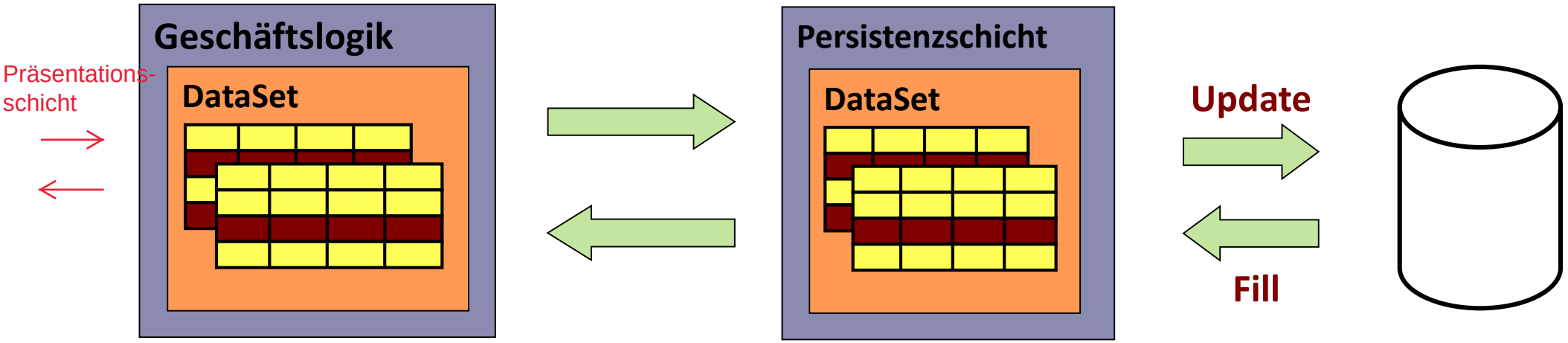
  - Beim Zurückschreiben in Datenbank wird *AcceptChanges* aufgerufen.

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

Austausch von Datasets

- Datasets sind für den Austausch von Daten zwischen *verschieden 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.

nix für die Projektarbeit



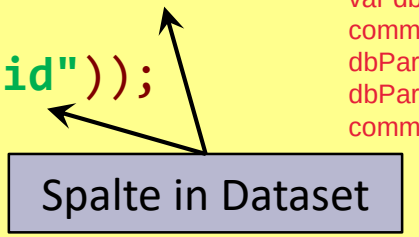
Daten-Adapter – Verbindung zur Datenquelle

- **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", SqlType.Double, 0, "price"));
updCmd.Parameters.Add(
  new SqlParameter("@id", SqlType.Integer, 0, "id"));
articleAdapter.UpdateCommand = updCmd;
articleAdapter.Update(shopDS.Article);
```



oder auch
var dbParam =
command.CreateParameter();
dbParam.ParameterName = p.Name;
dbParam.Value = p.Value;
command.Parameters.Add(dbParam);

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

pesimistic Locking (Formular mit Datensatz - beim Lesen des Datensatzes wird Datenbank der Datensatz gelockt - jemand anderes hat derweil keine Möglichkeit den Datensatz zu verändern - kann aber anschauen)
-Webwelt- 'niedergelockt' - zu viele Locks

- ADO.NET verwendet *optimistisches Locking*.
  - Datensätze werden beim/nach Befüllen des DataSets nicht gelockt.
  - Nur während des Synchronisierens mit der Datenbank wird gelockt.

kein Lock
beim zurückschreiben schau ich ob sich die Daten geändert haben - was soll passieren? - selbst implementieren

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

DataSets unterstützen das automatisch

- Sonst wird eine *DbConcurrencyException* geworfen.

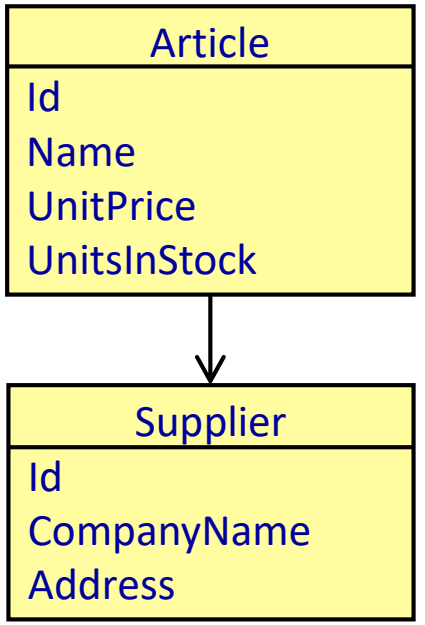
DataSets speichern Änderungen der Daten gelesenen Wert aus DB und aktuellen Wert (bzw. neuer Wert)

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

update nur dann wenn in db das steht wie beim lesen

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.



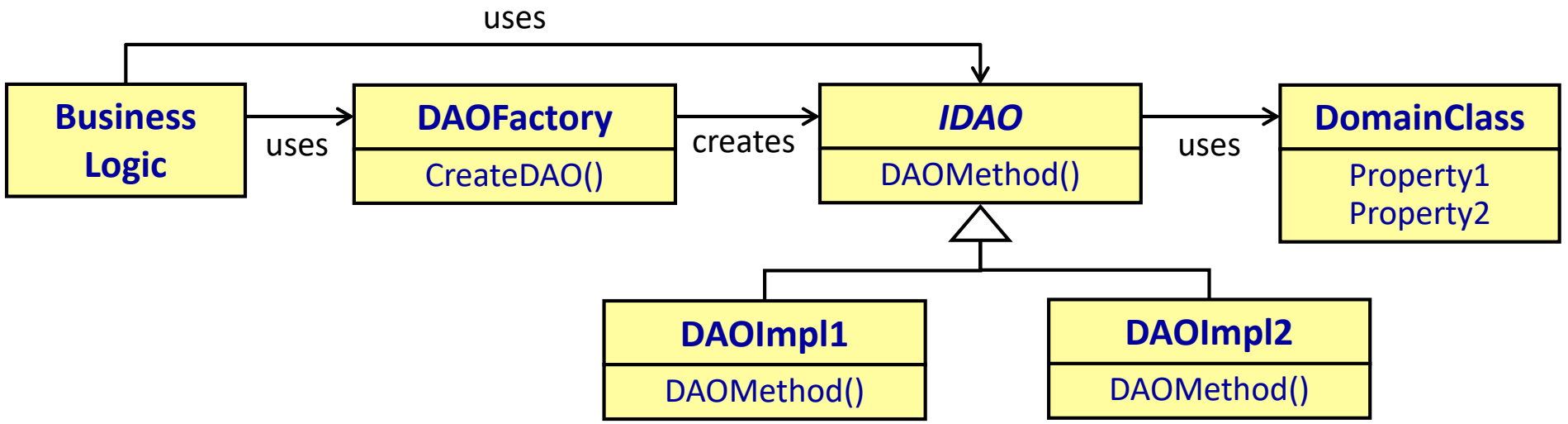
- Domänenklassen sind völlig technologieunabhängig.
- Die Manipulation der Daten wird in eigenen Datenzugriffsklassen (DAOs) durchgeführt.

Das DAO-Muster

Datenzugriffsoperation selber implementieren

handle mir die Operationen ein und muss sie imiiplementieren

- 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.



Domänenklassen/Datasets: Rolle in der Architektur

POCOs - plain old java objects??

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