# Geschäftsobjekte und ADO.NET

© 2003, Manfred Rätzmann

RÄTZMANN GmbH, Claudiusstr. 3, 10557 Berlin

mailto: m.raetzmann@raetzmann-gmbh.de

## Teil 1: Geschäftsobjekte generell

Mit ADO.NET hat Microsoft alle technischen Voraussetzungen für die strikte Trennung von Oberfläche, Verarbeitung und Datenhaltung geschaffen. Geschäftsobjekte (Business Objects) und Geschäftsobjekt-Manager sind ein zentraler Bestandteil einer flexiblen Softwarearchitektur. Im ersten Teil dieses Artikels wird ein Design Modell dafür vorgestellt. Im zweiten Teil soll dieses Modell dann in C# realisiert werden. Dieser Text ist in leicht abgewandelter Form als zweiteiliger Artikel ist in den Ausgaben 02.03 und 04.03 im „dot.net magazin“ erschienen.

Geschäftsobjekte - was ist das eigentlich und warum sind sie so wichtig? Ein Geschäftsobjekt repräsentiert ein Ding (oder einen Vorgang) aus der realen Welt, für die unser Programm erstellt wird. Da Software häufig dazu dient, Geschäftsprozesse zu automatisieren, hat sich der Ausdruck "Geschäfts"-objekt oder Business Object dafür etabliert.

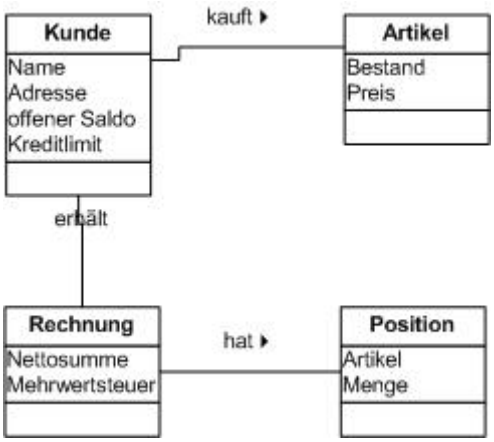


**Abb. 1: Ganz einfaches Modell mit Geschäftsobjekten**

In Abbildung 1 sehen Sie ein stark vereinfachtes Modell eines nicht unüblichen Geschäftsumfelds. Die darin enthaltenen Geschäftsobjekte sind "Kunde", "Artikel" und "Rechnung". Vor die Aufgabe gestellt, ein Programm zu schreiben, das die Rechnungsstellung übernimmt, wird sich mancher gestandene Entwickler nun vor seinen Rechner setzen, sein Lieblingsentwicklungswerkzeug aufrufen, ein neues Projekt anlegen und wahrscheinlich mit der Kundenstammverwaltung oder der Artikelstammverwaltung anfangen: Datenbanktabelle aufbauen, neues Formular erstellen, datengebundene Steuerelemente rein, Button zum Navigieren, Speichern und Löschen dazu, Click-Events ausprogrammiert - fertig! Vielleicht braucht das Formular noch die ein- oder andere Methode aber im Prinzip funktioniert das so, oder?

## Das Gute an Geschäftsobjekten

Geschäftsobjekte ermöglichen es, das in der Analyse entwickelte Geschäftsmodell für die zu erstellende Anwendung ohne Bruch zum Designmodell, Datenmodell und, wenn nötig, zum Implementationsmodell weiter zu entwickeln. Aus dem Kunden im Geschäftsmodell wird die Klasse "Kunde" im Designmodell und im Sourcecode. Ebenso werden wir ein Klasse "Artikel" und eine Klasse "Rechnung" im Sourcecode wiederfinden.

Geschäftsobjekte machen uns unabhängig von der Oberfläche. Der für die Anwendung wichtige Code steckt im Geschäftsobjekt, nicht in einer einzelnen Windows- oder Web-Form. Ein einzelnes Geschäftsobjekt kann einen

Web-Service bereitstellen, eine Windows-Form bedienen oder von einem Batch-Prozess genutzt werden. So kann die Oberfläche ausgetauscht werden ohne dass der Kern der Anwendung neu entwickelt werden müßte.

Für Programmteile, die nicht mit der Benutzeroberfläche interagieren, lassen sich automatisierte Unit-Tests aufbauen. Sobald die Benutzeroberfläche in's Spiel kommt wird die Testautomation erheblich schwieriger, wenn nicht sogar unmöglich. Auch Integrationstests oder Systemtests für größere Module und Subsysteme lassen sich viel einfacher automatisieren, wenn auf eine strikte Trennung zwischen Oberfläche, Verarbeitung und Datenhaltung geachtet wird wie in Abbildung 2 dargestellt.

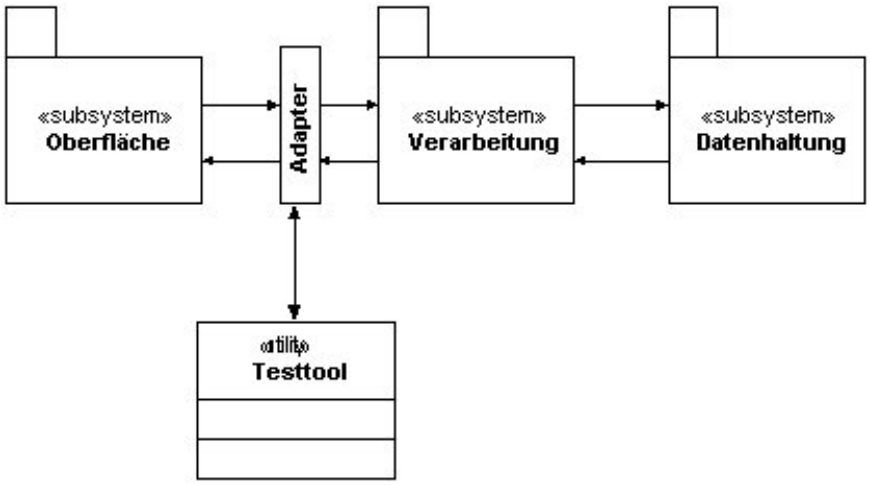


**Abb. 2: Adapter zur Steuerung und Aufzeichnung von Testabläufen**

Geschäftsobjekte machen uns auch unabhängig von der Datenhaltung. Durch Kapselung des Datenzugriffs in gesonderten Zugriffsklassen kann das Programm bei der Installation auf verschiedene Datenbanken eingestellt werden - einfach indem die Geschäftsobjekte für die Verwendung einer anderen Zugriffsklasse konfiguriert werden. Wenn erforderlich kann ein Wechsel der Datenbank sogar zur Laufzeit erfolgen.

Zu guter Letzt helfen Geschäftsobjekte dabei, den Code der Anwendung besser zu organisieren. Die Trennung von Oberfläche, Verarbeitung und Datenhaltung bietet das notwendige Ordnungsprinzip, die Verwendung von Geschäftsobjekten die logische Struktur.

## Geschäftsobjekte und Daten

Wenn Sie Datenbankdesigner sind, werden Sie bei dem kleinen Geschäftsmodell oben vielleicht gleich an ein Entity-Relationship Diagramm gedacht haben. In diesen Diagrammen werden Einheiten (Entitäten) aus dem modellierten Bereich zueinander in Beziehung (Relationship) gesetzt. Entity-Relationship Diagramme werden beim logischen Datenbankentwurf eingesetzt. Die Diagramme der Analysephase sind sehr häufig ebenfalls Entity-Relationship Diagramme, auch wenn sie in UML Notation als Klassendiagramme ausgeführt sind. Die bei der Analyse bereits ermittelten Daten werden als Klassenattribute angegeben. Als Datenbankdesigner werden Sie wenig Schwierigkeiten haben, das Geschäftsmodell oben in eine passende Tabellenstruktur umzusetzen.

Doch das Datenmodell zeigt nur die halbe Wahrheit. Die Entitäten des geschäftlichen Umfelds werden nicht nur durch ihre Attribute beschrieben - durch Daten also - sondern auch durch ihr Verhalten. Was liegt demnach näher als die in der Analyse gefundenen Geschäftsobjekte als Klassen zu realisieren, die ihre nicht flüchtigen Daten in Datenbanktabellen ablegen? Das Verhalten der Geschäftsobjekte schlägt sich in den Methoden dieser Klassen nieder.

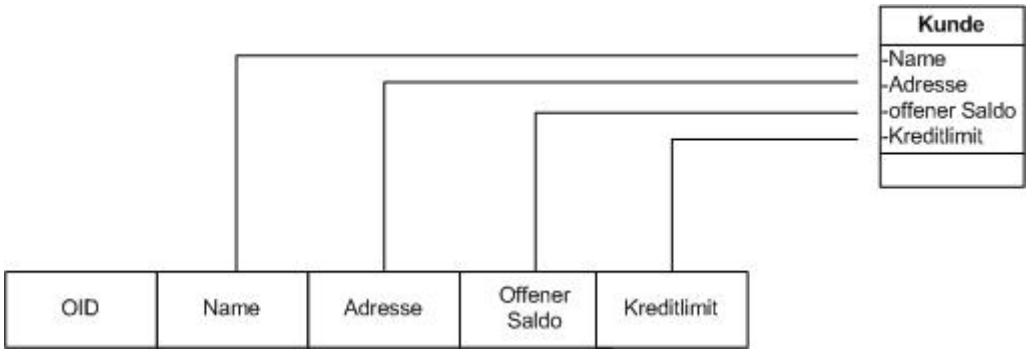


**Abb. 3: Einfache Projektion von Klassen auf Tabellen**

Die in Abbildung 3 dargestellte einfache Art der Projektion von Klassen auf Datenbanktabellen ist sicher nicht immer und überall anwendbar, aber sie trägt schon sehr weit. Ein einzelner Datensatz entspricht dabei einem Objekt der Klasse, ein Datenfeld im Datensatz entspricht einem nicht-flüchtigen (persistenten) Attribut der Klasse. Ein wenig aufwändiger wird die Sache, wenn man Vererbungsstrukturen auf Tabellen abbilden muss. Eine detaillierte Beschreibung der Vorgehensweise dabei finden Sie zum Beispiel unter [Rätzmann2000].

## Zuständigkeiten verteilen

Unabhängig von der Art der Datenhaltung vereinfachen Geschäftsobjekte beim Design das Verteilen von Zuständigkeiten. Alles das, wofür ein einzelnes Kundenobjekt zuständig ist, zum Beispiel Ermittlung und Rückgabe des Saldos seiner offenen Posten, wird als Methode des Geschäftsobjektes "Kunde" modelliert. Das Geschäftsobjekt "Rechnung" ist zuständig für die Berechnung der Gesamtsumme und Ermittlung des Mehrwertsteuerbetrages, das Geschäftsobjekt "Artikel" ist zuständig für die Reduzierung seines Bestands bei einer Lieferung und so weiter und so fort. Mit Geschäftsobjekten "normalisieren" Sie quasi Ihr Programm - auch das eine Analogie zum Datenbankdesign. Beim Datenbankdesign entfernen Sie durch Normalisierung unnötige und hinderliche Redundanzen. Beim Applikationsdesign vermeiden Sie redundanten Code durch die Verwendung von Geschäftsobjekten, Geschäftsobjekt-Managern und Dienstanbietern. Eine Faustregel beim Verteilen der Zuständigkeiten lautet:

1. Ein Geschäftsobjekt ist immer nur für sich selbst zuständig, ein Kunden-Objekt kann also den Saldo der eigenen offenen Posten ermitteln und auf Anforderung zurück geben. Es kann jedoch keine Umsatzstatistik für mehrere Kunden erstellen.

2. Alles, was mehrere Geschäftsobjekte des gleichen Typs betrifft, wird von den Geschäftsobjekt-Managern erledigt. Der KundenManager ist somit zuständig für kundenbezogene Auswertungen oder Mengenoperationen aber auch für das Erstellen neuer Kunden-Objekte, den Abruf gespeicherter Objekte und das Speichern geänderter Objekte.

3. Was nicht ein einzelnes Geschäftsobjekt oder eine Menge von gleichartigen Geschäftsobjekten betrifft, ist Sache eines spezialisierten Dienstanbieters. Ein Druckdienst-Anbieter könnte zum Beispiel seine Dienste allen Geschäftsobjekten anbieten, die eine IDruckbar Schnittstelle haben.

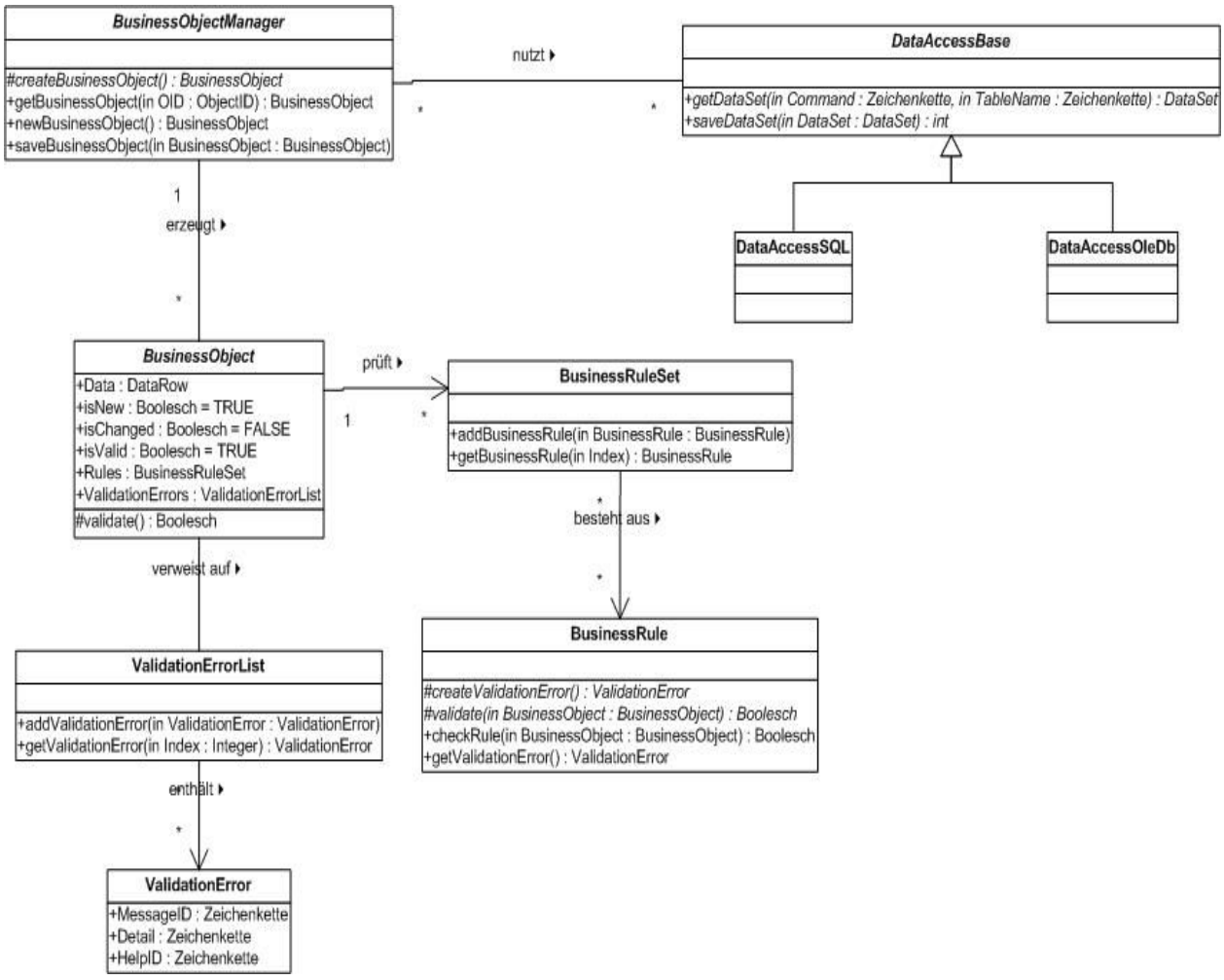


**Abb. 4: Designmodell Business Object Management**

## Ein Designmodell

Abbildung 4 zeigt ein mögliches Designmodell für Geschäftsobjekte, Geschäftsobjekt-Manager und weitere benötigte Hilfsklassen als Klassendiagramm in UML Notation. Dieses Modell erhebt nicht den Anspruch, das einzig mögliche oder richtige Modell zu sein. Das Grundkonzept des Designmodells ist einfach: Der Geschäftsobjekt-Manager (BusinessObjectManager) erzeugt die benötigten Geschäftsobjekte (BusinessObject) und nutzt zum Zugriff auf die Datenbank separate Datenzugriffsobjekte (DataAccessSQL oder DataAccessOleDB). Im Design Modell aus Abbildung 4 sind nur die öffentlichen Schnittstellen der einzelnen Klassen aufgeführt. Interne Klassenelemente sind nicht dargestellt.

Die BusinessObjectManager Klasse ist eine abstrakte Klasse, das heißt, aus dieser Klasse werden keine Objekte erstellt. Aus der BusinessObjectManager Klasse werden die konkreten Managerklassen abgeleitet wie KundenManager, ArtikelManager, RechnungsManager und so weiter. Das gleiche gilt für die BusinesObject Klasse. Auch diese dient als abstrakte Basisklasse zur Ableitung der konkreten Klassen für Kunden, Artikel und so weiter. Abbildung 5 zeigt dieses Ableitungsschema. Natürlich beschränken sich die konkreten Manager nicht auf die drei ererbten Operationen. Sie sind, wie oben beschrieben, Sammelpunkt für alle Aktionen, die mehr als ein Geschäftsobjekt des verwalteten Typs betreffen. Der KundenManager wäre zum Beispiel für eine Operation "erstelleUmsatzStatistik" zuständig. Diese Operation könnte dann ein DataSet zurück liefern, das von der Benutzeroberfläche in Form eines Berichts oder einer Grafik ausgegeben wird.

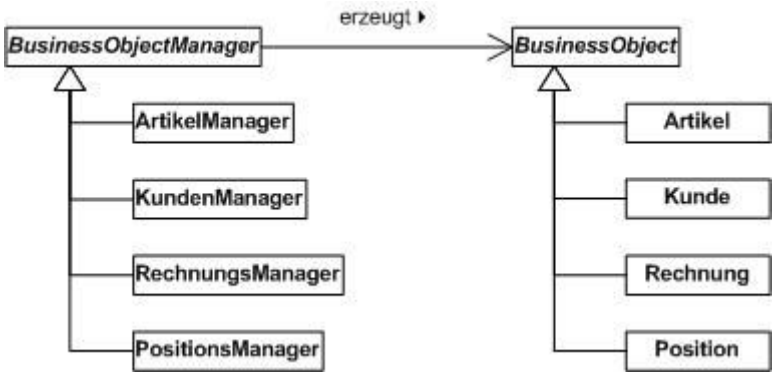


**Abb. 5: Die abgeleiteten konkreten Klassen**

BusinessObjectManager erzeugen BusinessObjects. KundenManager erzeugen Kunden-Objekte, ArtikelManager erzeugen Artikel-Objekte und RechnungsManager erzeugen selbstverständlich nur RechnungsObjekte. Da die Basisklasse aller BusinessObjectManager nicht wissen kann, welcher konkrete BusinessObject Typ durch ihren jeweiligen Abkömmling letztendlich erzeugt werden soll, muss diese Frage offen bleiben - sie kann erst durch die konkreten Manager selbst beantwortet werden. Das ist ein klassischer Fall für das "Factory Method" Muster [Gamma95]. Die Operation createBusinessObject wird in der Basisklasse lediglich deklariert, bleibt jedoch abstrakt. Die abgeleiteten Manager müssen die Methode ausformulieren, das heißt den jeweils passenden BusinessObject Typ erstellen und zurück liefern. Weitere Operationen der BusinessObjectManager Klasse können dann die so erzeugten BusinessObjects verwenden.

Die "getBusinessObject" Operation erwartet eine ObjectID als Input Parameter und liefert das BusinessObject mit dieser ID zurück. ObjectIDs sind eindeutige Kennzeichner für Objekte. Sie entsprechen den Primärschlüsseln in der Datenbank. ObjectIDs sollten künstliche Schlüssel sein. Natürliche Schlüssel wie zum Beispiel die Kunden-Nummer oder die Artikel-Nummer eignen sich nicht als ObjectID. Um vollständig datenunabhängig zu bleiben sollten auch nicht die Autonumber Felder der Datenbank verwendet werden. Am besten erstellen Sie ObjectIDs als GUID (Global Unique ID), auch wenn Ihnen das zunächst als Platzvergeudung erscheinen mag. Mit GUIDs lassen sich verteilte Systeme einfach handhaben (zumindest, was die ObjectIDs angeht) - wer weiß, ob aus Ihrem System nicht irgendwann auch ein verteiltes System wird.

"newBusinessObject" liefert ein leeres BusinessObject zurück. Weil "newBusinessObject" dazu intern die "createBusinessObject" Methode benutzt, wird natürlich der richtige Typ zurück geliefert (siehe oben). Der angegebene Rückgabetyp "BusinessObject" ist nur der kleinste gemeinsame Nenner und kann vom aufrufenden Programm in den konkreten Typ umgewandelt werden.

## Datenzugriff

Die "saveBusinessObject" Operation schließlich speichert das übergebene BusinessObject wieder in der Datenbank ab. Die BusinessObjectManager Klasse nutzt für alle Zugriffe auf die Datenbank ein Datenzugriffsobjekt, das aus einer DataAccess Klasse erstellt wird. Alle DataAccess Klassen werden von DataAccessBase abgeleitet. Die DataAccess Klassen sind datenbankspezifisch. Durch die Auslagerung des eigentlichen Datenzugriffs in DataAccess Klassen kann die tatsächlich benutzte Datenbank einfach konfigurierbar gehalten werden. Wenn nötig könnte die Datenbank sogar zur Laufzeit einfach umgeschaltet werden indem der BusinessObjectManager ein anderes DataAccess Objekt benutzt.

Die Kommunikation mit der Datenbank läuft im Wesentlichen über die Operationen "getDataSet" und "saveDataSet" der DataAccess Klasse. Wenn ein einzelnes BusinessObject geladen oder gespeichert wird, enthält der DataSet nur eine Tabelle mit einem Satz. Der BusinessObjectManager kann aber bei Bedarf auch DataSets mit mehreren Tabellen und Datensätzen abrufen und als Ergebnis eigener Operationen zurück liefern. Die DataAccess Klassen- und Methodennamen in diesem Modell habe ich mit freundlicher Genehmigung des Autors aus [McNeish02] übernommen.

BusinessObjects besitzen eine Eigenschaft "Data". Hinter "Data" verbirgt sich ein DataRow Objekt, also eine Zeile einer Tabelle aus einem DataSet. Über einKunde.Data["Name"] kann man also auf den Namen des Kunden-Objekts "einKunde" zugreifen, wenn der direkte Zugriff auf die "Data" Eigenschaft zugelassen wird. Alternativ kann man die Attribute des Geschäftsobjektes wie "Name", Adresse" und so weiter als Eigenschaften veröffentlichen und intern auf die Felder des DataRow Objektes zugreifen.

Der Status des BusinessObject kann über verschiedene boolesche Werte abgeprüft werden. isNew gibt an, ob das BusinessObject bereits in der Datenbank gespeichert ist (FALSE) oder nicht (TRUE), isChanged gibt an, ob

das BusinessObject seit dem Abruf aus der Datenbank hier geändert wurde. isValid gibt zurück, ob das BusinesObjects in seinem aktuellen Zustand den hinterlegten Geschäftsregeln entspricht.

## Geschäftsregeln überprüfen

Zur Validierung wird von der BusinessObject Klasse eine geschützte "validate" Methode angeboten, die im Original einfach ein TRUE zurück gibt. Wenn es eine feststehende Menge von Geschäftsregeln gibt, denen das BusinessObject zu jedem Zeitpunkt entsprechen muss, kann die "validate" Methode von den abgeleiteten konkreten BusinessObject Klassen überschrieben werden um diese Regeln abzuprüfen. Zusätzlich kann der Eigenschaft "Rules" ein Satz von Regeln (BusinessRuleSet) zugewiesen werden, der aus beliebig vielen einzelnen Geschäftsregeln (BusinessRule) bestehen kann. Damit ist zusätzlich zur fixen Validierung über die "validate" Methode eine dynamische Validierung möglich, die vom Status des BusinessObjects innerhalb eines Prozessablaufs abhängig gemacht werden kann. Ein BusinessRule Objekt besteht im wesentlichen aus den beiden geschützten Methoden "validate" und "createValidationError". Die veröffentlichte Operation "checkRule" wird vom BusinessObject aufgerufen um eine einzelne Regel zu überprüfen. Dabei übergibt das BusinessObject sich selbst als Prüfobjekt. Die Aufrufsequenz dieses Ablaufs ist in Abbildung 6 dargestellt. Die Rückgaben habe ich dabei der Einfachheit halber weggelassen.

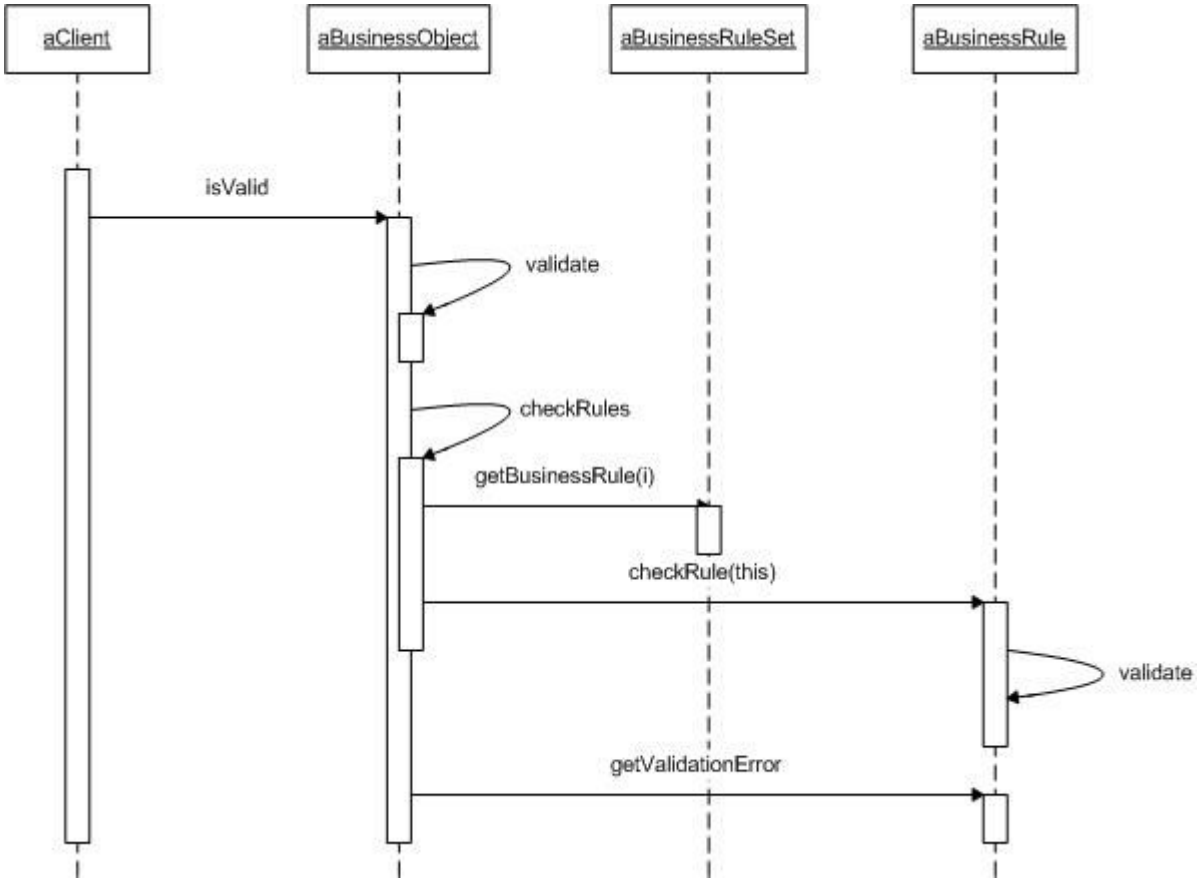


**Abb. 6: Sequenz der Geschäftsregelüberprüfung**

Wenn ein Client Objekt die "isValid" Eigenschaft eines BusinessObjects abprüft wird zunächst die fixe "validate" Methode ausgeführt und danach für jede Geschäftsregel aus dem angeschlossenen BusinessRuleSet die "checkRule" Methode aufgerufen. Alle "validate" Methoden geben ein FALSE zurück, falls die Regel gebrochen wurde. Die fixe "validate" Methode des BusinessObjects sorgt in diesem Fall selbst dafür, dass für jede gebrochene fixe Regel ein ValidationError Objekt den ValidationErrors hinzugefügt wird. Meldet ein BusinessRule Objekt beim Aufruf von "checkRule" ein FALSE an das aufrufende BusinessObject zurück, so ruft das BusinessObject anschließend die "getValidationError" Methode der BusinessRule auf und legt das zurück erhaltene ValidationError Objekt in der ValidationErrorList ab. Über die Eigenschaft "ValidationErrors" kann das aufrufende Client Objekt die bei der Validierung aufgetretenen Fehler auswerten.

## Teil 2: Geschäftsobjekte in C#

Das grundsätzliche Design einer 3-Schichten Anwendung ist unabhängig von der eingesetzten Programmiersprache und wurde oben an Hand eines Designmodells erläutert. Jetzt wollen wir dieses Designmodell als C# Applikation realisieren. Abbildung 1 zeigt einen Ausschnitt des erweiterten Modells für die Implementation in C#. Die BusinessObjectManager Klasse nutzt ein DataAccess Object, das aus der abstrakten Klasse DataAccessBase abgeleitet ist. Im Beispiel wird ein DataAccessSQL Objekt benutzt, das auf den Zugriff auf eine SQL Server Datenbank spezialisiert ist. Wenn statt dessen ein DataAccessOleDB Objekt genutzt würde, könnten die Daten aus einer beliebigen OLE DB fähigen Datenbank stammen, ohne dass der BusinessObjectManager davon etwas merken würde oder darauf umgestellt werden müßte. Der Wechsel zwischen Datenbanken kann, wie wir gleich sehen werden, ohne Neukompilierung und bei Bedarf sogar zur Laufzeit erfolgen.

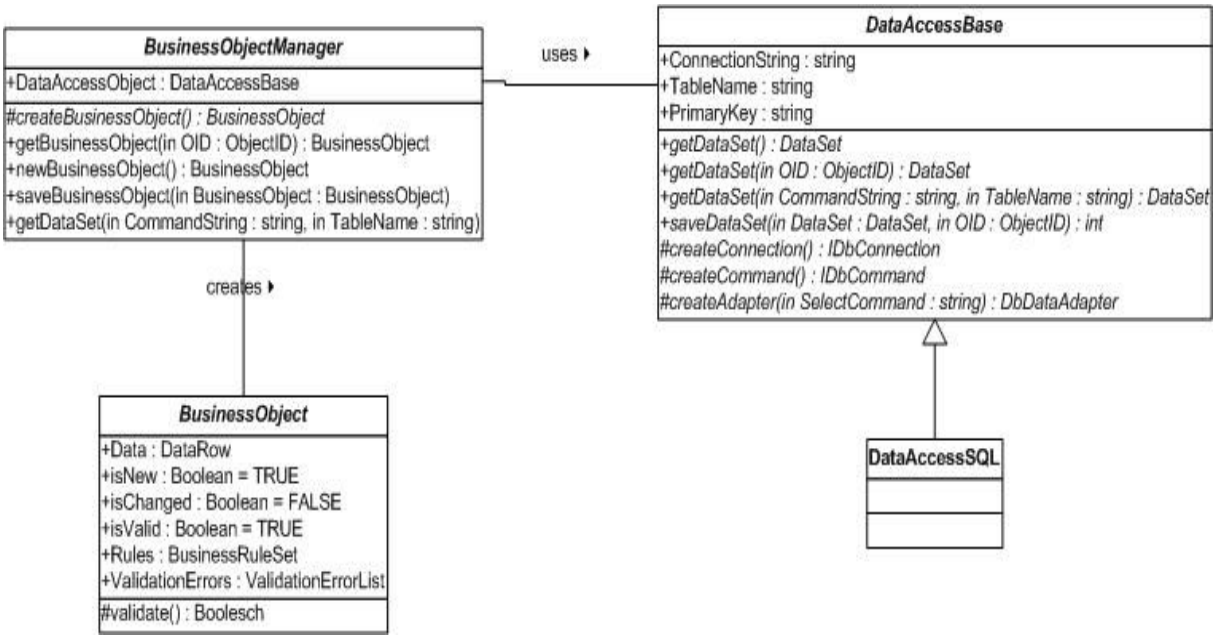


**Abb. 1: Ausschnitt aus C# Implementationsmodell**

## Die Basisklassen

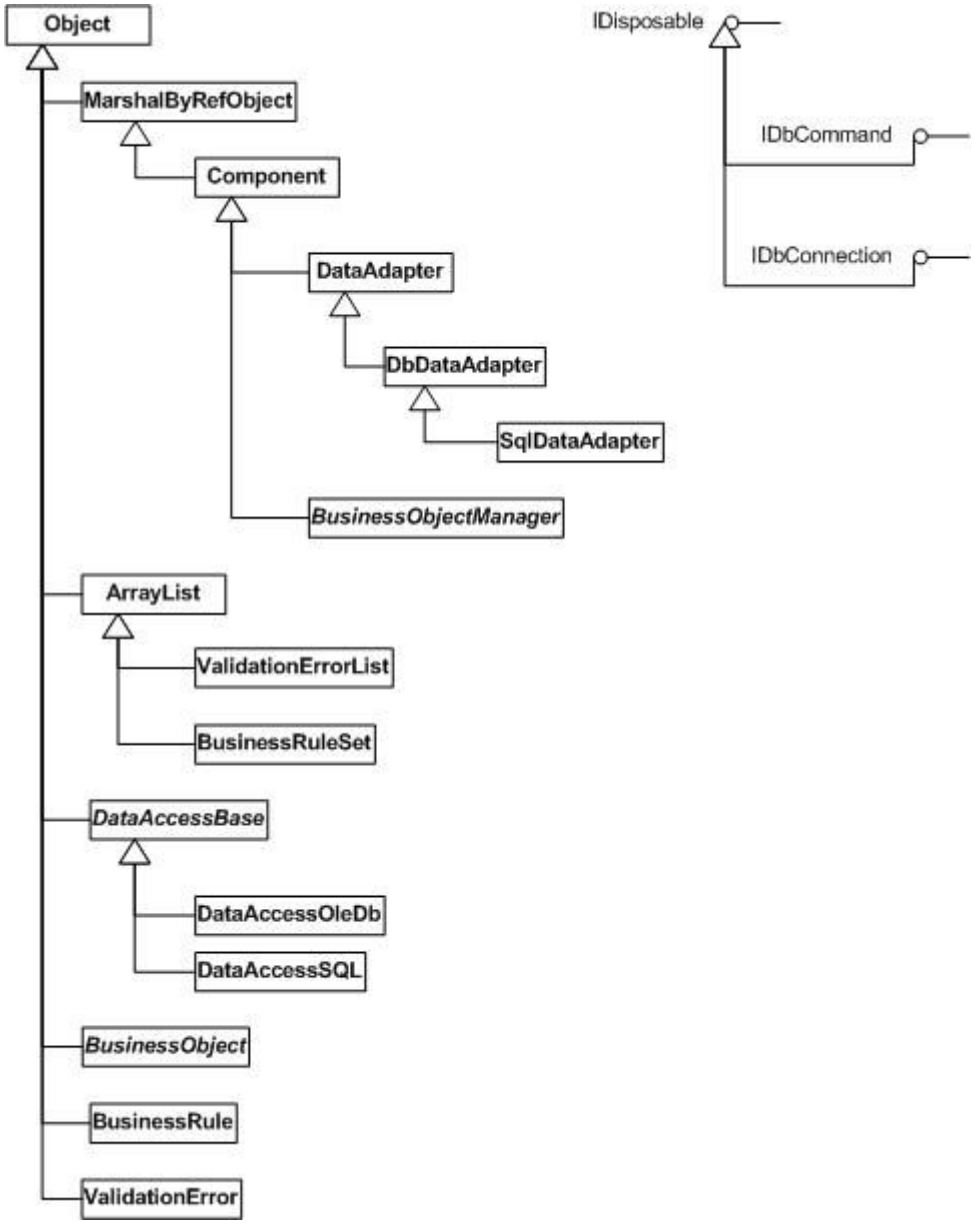


**Abb. 2: Basisklassen des BOManagements**

Abbildung 2 zeigt die Basisklassen des .NET Frameworks, aus denen die Klassen des Designmodells abgeleitet wurden. Die BusinessObjectManager Klasse wurde von Component abgeleitet und kann damit bei Bedarf in die Toolbox aufgenommen werden. Die BusinessObject Basisklasse wurde direkt aus Object abgeleitet, da Geschäftsobjekte immer von einem entsprechenden Manager erzeugt und nicht als eigenständige Komponente verwendet werden sollen.

Die Auflistungen von Geschäftsregeln (BusinesRuleSet) und Validierungsfehlern (ValidationErrorList) wurden aus der ArrayList Klasse abgeleitet, da diese einfach zu verwalten ist und dynamisch erweitert werden kann. Wer weiß schon im voraus, wie viele Geschäftsregeln und Validierungsfehler ihn erwarten?

Der Vollständigkeit halber ist in Abbildung 2 auch der Vererbungsbaum der in der DataAccessBase Klasse benutzten Schnittstellen angegeben.

## Die Klasse DataAccessBase

Listing 1 zeigt die Implementierung der abstrakten Klasse DataAccessBase, aus der die spezialisierten Zugriffsklassen für SQL Server oder OLE DB Datenbanken abgeleitet werden. Der Code ist auf das Notwendigste reduziert. Er muss im richtigen Leben natürlich noch durch Ausnahmebehandlung, den ein- oder anderen Kommentar und ähnliches ergänzt werden.

```
public abstract class DataAccessBase
{
  protected string _ConnectionString;
  public string ConnectionString
  {
    get {return _ConnectionString;}
    set {_ConnectionString = value;}
  }
  protected string _TableName;
  public string TableName
  {
    get {return _TableName;}
    set {_TableName = value;}
  }

  protected string _PrimaryKey;
  public string PrimaryKey
  {
    get {return _PrimaryKey;}
    set {_PrimaryKey = value;}
  }

  protected abstract IDbConnection createConnection();
  protected abstract IDbCommand createCommand(string CommandString,
IDbConnection Connection);
  protected abstract DbDataAdapter createAdapter(string SelectCommand);
  public abstract DataSet getDataSet();
  public abstract DataSet getDataSet(string OID);
  public abstract DataSet getDataSet(string CommandString, string
TableName);
  public abstract int saveDataSet(DataSet DataSet, string OID);
}
```

**Listing 1: Die generelle Zugriffsklasse DataAccessBase**

DataAccessBase ist eine abstrakte Klasse. Aus dieser Klasse können also keine Objekte erstellt werden. Abstrakte Klassen dienen als Elternklasse bei der Ableitung spezialisierter konkreter Klassen. Die Klasse DataAccessBase ist direkt von der Klasse "Object" abgeleitet. Aus diesem Grund brauchen wir bei der Definition von DataAccessBase selbst keine Elternklasse anzugeben.

Innerhalb von DataAccessBase werden zunächst drei Eigenschaften der Klasse beschrieben. Eigenschaften sind immer mit einer get und/oder einer set Accessor-Methode verknüpft. Der ConnectionString beschreibt die Verbindung zur Datenbank und kann beim Programmstart zum Beispiel aus einer INI Datei gelesen oder zur Laufzeit zusammengebaut werden. Damit ist die später tatsächlich genutzte Datenbank bereits variabel addressierbar. Um auch den Zugriff auf die einzelne Tabelle innerhalb der Datenbank variabel zu halten, benötigen wir noch den Namen der Tabelle und deren Primärschlüssel. Dazu werden die Eigenschaften "TableName" und "PrimaryKey" deklariert.

Da wir den Typ der später benutzten Datenbank ebenfalls als unbekannt voraussetzen, reicht es aber nicht aus, lediglich den ConnectionString, Tabellen-Namen und den Primärschlüssel variabel zu halten. Wenn wir später einen DataSet füllen, müssen wir die zur Datenbank passenden konkreten Connection, Adapter und Command Klassen verwenden. Deshalb können wir hier in der DataAccessBase Klasse noch keine Methoden zum Zugriff auf die Daten ausprogrammieren. Alle Methoden selbst sind daher als "abstract" deklariert, das heißt, es folgt in dieser Klasse kein Methodencode sondern die abgeleiteten spezialisierten Klassen müssen den Methodencode bereit stellen. In der Deklaration der Methoden createConnection(), createCommand() und createAdapter() müssen wir aus dem gleichen Grund auf die Rückgabe eines generellen Typs ausweichen, da die konkret zu verwendenden Typen erst von den abgeleiteten Klassen festgelegt werden. Die createConnection() Methode der konkreten DataAccessSQL Klasse muss also ein SqlConnection Objekt zurück liefern, während die gleiche

Methode der DataAccessOleDB Klasse ein OleDBConnection Objekt zurückgeben soll. Alle Datenbank-Connection Klassen müssen die Schnittstelle IDbConnection bereit stellen. Ebenso müssen alle Datenbank-Command Klassen die IDbCommand Schnittstelle anbieten. Deshalb benutzen wir diese beiden Schnittstellen als Rückgabetypen für createConnection() und createCommand(). In den später abzuleitenden Klassen werden wir den gelieferten Typ in den konkret benötigten Typ umwandeln. Ähnlich verhält es sich bei den benötigten DataAdapter Klassen. Hier bildet DbDataAdapter die Elternklasse für alle Datenzugriffs-Adapter.

Zum Zugriff auf die Daten wird eine mehrfach überladene Methode namens getDataSet angeboten. Wie der Name schon sagt, wird bei allen Aufrufen ein DataSet zurück geliefert. Wenn beim Aufruf keine weiteren Parameter übergeben werden, enthält der zurück gelieferte DataSet die Daten eines neuen - d.h. noch nicht in der Datenbank gespeicherten - Geschäftsobjekts. Wenn die Daten eines bereits gespeicherten Geschäftsobjekts abgerufen werden sollen, muss beim Aufruf von getDataSet die Object-ID (OID) des gewünschten Geschäftsobjektes übergeben werden. Wenn eine Liste von gespeicherten Geschäftsobjekten benötigt wird, kann getDataSet mit einem passenden SQL SELECT Kommando (CommandString) aufgerufen werden. Zusätzlich wird hierbei noch der Tabellenname (TableName) angegeben, unter dem der Ergebniscursor im DataSet gespeichert werden soll.

Zum Aktualisieren der Daten eines Geschäftsobjektes - also für INSERT, UPDATE oder DELETE reicht eine allgemeine saveDataSet Methode. Aktualisiert werden in dieser Version immer nur die Daten eines einzelnen Geschäftsobjektes, dessen OID beim Aufruf der saveDataSet Methode als zweiter Parameter mit dem DataSet übergeben wird.

## Der konkrete Datenzugriff

Schauen wir uns als nächstes den konkreten Zugriff auf eine SQL Server Datenbank an. Listing 2 enthält den Code der aus DataAccessBase abgeleiteten Klasse DataAccessSQL. Auch hier nur das Notwendigste und ein paar Kommentare zur besseren Lesbarkeit des Codes.

```
public class DataAccessSQL : MR.BOManagement.DataAccessBase
{
   /// createCommand
protected override IDbCommand createCommand(string CommandString,
IDbConnection Connection)
   {
      return new SqlCommand(CommandString, (SqlConnection)Connection);
   }

   /// createConnection
   protected override IDbConnection createConnection()
   {
      return new SqlConnection(this.ConnectionString);
   }

   /// createAdapter
   protected override DbDataAdapter createAdapter(string SelectCommand)
   {
      IDbConnection _Connection = createConnection();
      IDbCommand _Command = createCommand(SelectCommand, _Connection);
      SqlDataAdapter _DataAdapter = new
SqlDataAdapter((SqlCommand)_Command);
      return _DataAdapter;
   }

   /// getDataSet(CommandString,TableName)
   public override DataSet getDataSet(string CommandString, string
TableName)
   {
      /// Create the adapter with the given SELECT command
      SqlDataAdapter _DataAdapter =
(SqlDataAdapter)createAdapter(CommandString);

      /// Create and fill the DataSet
```

```
      DataSet _DataSet = new DataSet();
      _DataAdapter.Fill(_DataSet, TableName);

      return _DataSet;
   }

   /// getDataSet(OID)
   public override DataSet getDataSet(string OID)
   {
      /// Create the adapter with the single-object SELECT command
      string _SelectCommand = "SELECT * FROM "+_TableName
         + " WHERE ("+_PrimaryKey+"='"+OID+"')";

      SqlDataAdapter _DataAdapter =
(SqlDataAdapter)createAdapter(_SelectCommand);
      DataSet _DataSet = new DataSet();
      _DataAdapter.Fill(_DataSet, _TableName);

      return _DataSet;
   }

   /// getDataSet()
   public override DataSet getDataSet()
   {
      /// Create the adapter for the empty table
      string _SelectCommand = "SELECT * FROM "+_TableName;

      SqlDataAdapter _DataAdapter =
(SqlDataAdapter)createAdapter(_SelectCommand);

      DataSet _DataSet = new DataSet();
      _DataAdapter.FillSchema(_DataSet,SchemaType.Mapped,_TableName);

      DataRow _Row = _DataSet.Tables[0].NewRow();
      _Row[_PrimaryKey] = System.Guid.NewGuid();

      _DataSet.Tables[0].Rows.Add(_Row);

      return _DataSet;

   }

   /// saveDataSet
   public override int saveDataSet(DataSet dataSet, string OID)
   {

      /// Create the adapter with the single-object SELECT command
      string _SelectCommand = "SELECT * FROM "+_TableName
         + " WHERE ("+_PrimaryKey+"='"+OID+"')";

      SqlDataAdapter _DataAdapter =
(SqlDataAdapter)createAdapter(_SelectCommand);

      SqlCommandBuilder _CommandBuilder = new
SqlCommandBuilder(_DataAdapter);
      _DataAdapter.DeleteCommand = _CommandBuilder.GetDeleteCommand();
      _DataAdapter.UpdateCommand = _CommandBuilder.GetUpdateCommand();
      _DataAdapter.InsertCommand = _CommandBuilder.GetInsertCommand();

      // Update the data of the DataSet
      int _RowsUpdated = _DataAdapter.Update(dataSet, _TableName);

      return _RowsUpdated;
   }
```

```
    }
```

**Listing 2: Die konkrete Zugriffsklasse DataAccessSQL**

In den verschiedenen Ausprägungen der getDataSet Methode wird zunächst ein DataAdapter erzeugt. Die spezialisierten DataAdapter für SQL-Server und OleDb Zugriff bieten jeweils einen Konstruktor, der das benötigte SQL-SELECT Statement als Parameter übernehmen kann. Da ein solcher Konstruktor für die Ableitung von neuen DataAdapter-Klassen aus DbDataAdapter ausdrücklich empfohlen wird, können wir davon ausgehen, dass auch auf andere Datenbanken spezialisierte DataAdapter diesen Konstruktor anbieten werden.

Wenn der benötigte DataAdapter zur Verfügung steht, können wir dessen Fill Methode aufrufen, um den DataSet mit Daten zu füllen. Den Namen für die zu füllende Tabelle nehmen wir bei einem Einzelobjekt-Zugriff aus dem Eigenschafts-Wert _TableName, beim Abruf einer Objektliste wird der Zieltabellenname als Parameter übergeben.

Wenn getDataSet() ohne Angabe von Parametern aufgerufen wird, soll ein DataSet mit einer Tabelle zurück gegeben werden, die einen nicht in der Datenbank gespeicherten neuen Datensatz enthält. Einen solchen Satz können wir natürlich nicht über ein SQL-SELECT mit anschließendem Aufruf der Fill Methode abrufen, da der Satz ja noch nicht vorhanden ist. Für solche Situationen bieten DataAdapter Klassen eine FillSchema Methode an. Damit kann eine leere Zieltabelle erzeugt werden, deren Spaltendefinitionen und Datentypen dem Ergebniscursor des SQL-SELECT Statement entsprechen. Wenn wir einer solchen Tabelle mit NewRow eine neue Zeile hinzufügen haben wir den benötigten leeren Datensatz für das neue Geschäftsobjekt erzeugt.

Abschließend füllen wir noch das Primärschlüsselfeld (die Objekt-ID) mit einem eindeutigen Wert, in diesem Fall mit einer neuen GUID. Diesen Vorgang in die Datenzugriffsklasse auszulagern hat den entscheidenden Vorteil, dass die Anwendung selbst nicht zu wissen braucht, wie eine Objekt-ID gebildet wird. Festgelegt ist lediglich der Datentyp "string" und die Tatsache dass jedes Objekt eine eindeutige ID hat, die völlig unabhängig von den Objektdaten ist (Existence-based Identity). Was in diesem String abgelegt wird und wie die benötigten eindeutigen Werte erstellt werden, muss die Anwendung nicht interessieren.

Für saveDataSet wird ebenfalls ein DataAdapter mit einem single-object SELECT Kommando benötigt. Das SELECT Kommando reicht aber nicht aus, wir brauchen jetzt auch die tabellenspezifischen DELETE, UPDATE und INSERT Kommandos. Diese müssen aber nicht vom Entwickler selbst erstellt werden, die Aufgabe nimmt uns ein SqlCommandBuilder ab. Dem Konstruktor des SqlCommandBuilders übergeben wir den aktuellen DataAdapter. Damit kennt der SqlCommandBuilder schon die Struktur des zu sichernden Cursors und wir können die passenden Kommandos über GetDeleteCommand(), GetUpdateCommand() und GetInsertCommand() abrufen. Wenn die Kommandos zur Verfügung stehen, rufen wir die Update Methode des DataAdapter Objekts auf. Die Update Methode prüft den aktuellen Inhalt des übergebenen DataSet und entscheidet selbst, ob Daten in der Datenbank zu löschen, neu anzulegen oder zu aktualisieren sind.

## BusinessObject und BusinessObjectManager

Die BusinessObject Klasse ist schnell erläutert, da sie lediglich zwei Eigenschaften deklariert. Die erste Eigenschaft heißt Data und ermöglicht den Zugriff auf die DataRow, die die Daten des Geschäftsobjekts enthält. Die zweite Eigenschaft heißt isNew und ist lediglich ein Flag, das aussagt, ob das Geschäftsobjekt bereits gespeichert ist oder nicht. Die Klassenvariablen dazu sind als "internal" gekennzeichnet. Dadurch kann die BusinessManager Klasse direkt auf die Klassenvariablen zugreifen und diese setzen während alle anderen Klassen außerhalb der BOManagement Assembly nur lesend auf die Eigenschaft zugreifen können. Um diesen Artikel nicht unnötig lang werden zu lassen haben wir auf die Realisierung der Validierungsmethoden (siehe Design Modell) verzichtet.

```
public abstract class BusinessObject
{

   internal DataRow _Data = null;
   public DataRow Data
   {
get {return _Data;}
   }

   internal Boolean _isNew = true;
```

```
    public Boolean isNew
    {
       get {return _isNew;}
    }

}
```

**Listing 3: Die abstrakte BusinessObject Klasse**

Die abstrakte BusinessObjectManager Klasse ist nur wenig aufwändiger. Listing 4 zeigt die Klassendefinition. Auch hier wird zunächst eine Eigenschaft deklariert, die zur Laufzeit mit einer Referenz auf das zu benutzende DataAccessObject bestückt wird. Da wir in der generell formulierten Manager Klasse die konkret verwaltete BusinessObject-Klasse nicht kennen, können wir die Methode createBusinessObject hier nicht ausformulieren. Das müssen wir den konkreten Managerklassen überlassen. Deshalb wird diese Methode hier lediglich als abstract deklariert.

Alle anderen generell benötigten Methoden können jedoch schon hier in der Mutter aller BusinessObjectManager ausprogrammiert werden. Die getBusinessObject-Methode ermöglicht den Abruf eines BusinessObject über dessen Object-ID, newBusinessObject erstellt und initialisiert ein neues BusinessObject, saveBusinessObject speichert ein neues oder geändertes BusinessObject in der Datenbank. Außerdem gibt es auch hier eine getDataSet Methode, mit dem Listen gespeicherter Geschäftsobjekte entsprechend dem in CommandString übergebenen SQL SELECT Kommando abgerufen werden können. Damit haben wir alles zusammen, was wir für eine erste kleine Beispielapplikation brauchen.

```
public abstract class BusinessObjectManager :
System.ComponentModel.Component
{
   private DataAccessBase _DataAccessObject;
   public DataAccessBase DataAccessObject
   {
      get {return _DataAccessObject;}
      set {_DataAccessObject = value;}
   }

   protected abstract BusinessObject createBusinessObject();

   public BusinessObject getBusinessObject(string OID)
   {
      BusinessObject BO = createBusinessObject();

      DataSet _BODataSet = _DataAccessObject.getDataSet(OID);
      BO._Data = _BODataSet.Tables[0].Rows[0];
      BO._isNew = false;
      return BO;
   }

   public BusinessObject newBusinessObject()
   {
      BusinessObject BO = createBusinessObject();
      DataSet _BODataSet = _DataAccessObject.getDataSet();
      BO._Data = _BODataSet.Tables[0].Rows[0];
      return BO;
   }

   public void saveBusinessObject(BusinessObject BO)
   {
      string _OID = BO.Data[_DataAccessObject.PrimaryKey].ToString();
      DataSet _BODataSet = BO.Data.Table.DataSet;

      _DataAccessObject.saveDataSet(_BODataSet,_OID);
      BO._isNew = false;
   }

   public DataSet getDataSet(string CommandString, string TableName)
   {
      return _DataAccessObject.getDataSet(CommandString,TableName);
```

```
    }
}
```

**Listing 4: Die abstrakte BusinessObjectManager Klasse**

## Eine Beispielapplikation

Als Beispiel für die Realisierung einer Applikation mit Geschäftsobjekten und Geschäftsobjekt-Managern habe ich eine einfache Zeiterfassung gewählt. Abbildung 3 zeigt das Geschäftsmodell der Beispielapplikation. Mit diesem Programm werden Zeiten festgehalten, die für Aktivitäten im Rahmen eines Projektes verwandt werden. Eine Aktivität beginnt zum Zeitpunkt "Start" und endet bei Begin der nächsten Aktivität. So braucht kein Timer mitzulaufen und die Zeiterfassung kann auch dezentral verwendet werden. Einzige Bedingung ist, dass immer eine Schlußaktivität wie zum Beispiel "Feierabend" erfasst wird, die dann bis zum nächsten Arbeitsbeginn gilt.

Die angefallenen Aktivitäten werden über Buchungssätze (Booking) mit Konten (Account) verknüpft, über die die Aktivität abzurechnen ist.

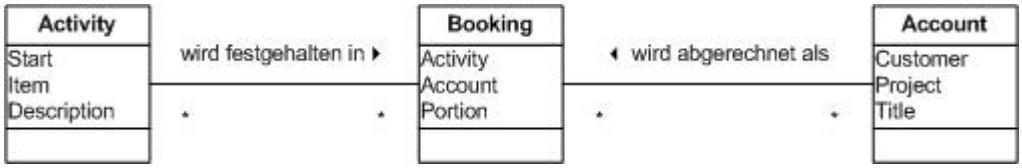


**Abb. 3: Geschäftsmodell der Beispielapplikation**

Als Beispiel dient hier die Verwaltung der Accounts. Dafür leiten wir zunächst mal eine Account Klasse aus der abstrakten Klasse BusinessObject ab. Diese Account Klasse können wir später nach Belieben mit dem Verhalten versehen, das ein Account-Objekt anbieten sollte. Hier reicht uns zunächst mal, dass es diese Klasse gibt und wir also Account-Objekte erstellen und speichern können. Diese Aufgabe übernimmt der AccountManager, den wir aus BusinessObjectManager ableiten wie in Listing 5 dargestellt.

```
public class AccountManager : BusinessObjectManager
{
   public AccountManager()
   {
      // initialize the data access object
      this.DataAccessObject = new DataAccessSQL();
      this.DataAccessObject.ConnectionString =
   "data source=MyHost\\VSdotNET;initial catalog=ActiveTimes;integrated
security=SSPI";

      // used to build the SQL statement
      this.DataAccessObject.TableName = "Accounts";
      this.DataAccessObject.PrimaryKey = "OID";
   }
   protected override BusinessObject createBusinessObject()
   {
      return new Account();
   }
}
```

**Listing 5: Der AccountManager**

Im Konstruktor legen wir die Eigenschaften des AccountManagers fest. Zunächst wird ein DataAccessObject zugewiesen, in diesem Fall ist es hart verdrahtet ein DataAccessSQL Objekt. Außerdem wird der ConnectionString beschrieben, der zur Verbindung mit der Datenbank gebaucht wird. Im richtigen Leben werden diese beiden Informationen natürlich in den Konfigurationsdaten der Applikation hinterlegt. Damit ist es möglich, die zu benutzende Datenbank von außen festzulegen.

Des weiteren muss der AccountManager noch wissen, wie die Tabelle in der Datenbank heißt, in der die Daten gespeichert werden sollen. Die Tabelle könnte natürlich in Wirklichkeit auch ein Datenbank-View sein, wenn Sie aus irgendwelchen Gründen die Daten eines Objektes auf mehrere Tabellen verteilen wollen. Als letzte Angabe im Konstruktor wird der Name des Primärschlüssels eingetragen.

Die einzige Fähigkeit, die wir dem neuen AccountManager beibringen müssen ist, Geschäftsobjekte des von ihm verwalteten Typs zu erzeugen. Dazu überschreiben wir die Factory-Method createBusinessObject und liefern bei

Aufruf ein neues Account Objekt zurück. Das war's. Jetzt können wir aus Windows- oder Web-Forms, aus anderen Programmteilen oder wo auch immer, Account Objekte erzeugen, speichern und wieder abrufen.

Versuchen wir es mal aus einer Windows-Form heraus. Abbildung 4 zeigt ein Testformular dazu im Designer, Listing 6 zeigt die wichtigsten Teile des Formularcodes dazu.

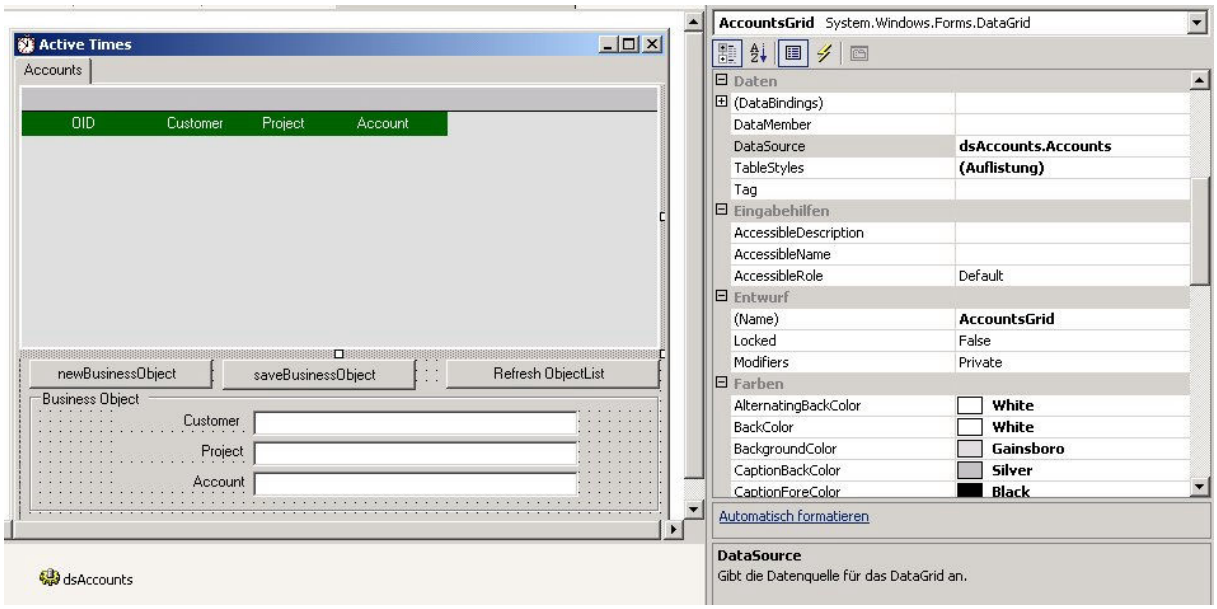


**Abb. 4: Das Beispielformular im Designer**

Zur Anzeige der Liste im Grid benutzen wir einen typisierten DataSet dsAccounts, der die Struktur der Accounts Tabelle bereit stellt. Damit müssen wir auch bei der Benutzung von Managerklassen nicht auf so schöne Dinge wie das DataBinding beim Grid verzichten. Allerdings wird der DataSet dsAccounts niemals direkt mit einer Datenquelle verbunden. Zum Aktualisieren der Account-Liste rufen wir stattdessen über den AccountManager _AccountMan die benötigten Daten ab und kopieren diese mit Hilfe der Merge Methode in den typisierten DataSet dsAccounts. Das an dsAccounts gebundene Grid merkt dann, dass neue Daten angekommen sind und zeigt diese an. Mehr als im Event Handler "AktualisierenButton_Click" von Listing 6 dargestellt ist dazu nicht nötig.

Im newButton_Click machen wir uns die Sache noch einfacher indem wir dem AccountManager lediglich mitteilen, dass wir ein neues Account Objekt benötigen und anschließend die displayBO Methode der Form zur Ausgabe der Daten auffordern. Beim Speichern im saveButton_Click aktualisieren wir die Daten des Account-Objektes mit den Werten der zugehörigen Textbox Controls und lassen den AccountManager das Objekt anschließend speichern. Wenn wir im Grid eine andere Zelle anwählen tritt das Ereignis AccountsGrid_CurrentCellChanged ein. In diesem Fall laden wir das in der Zeile dargestellte Geschäftsobjekt aus der Datenbank und zeigen es im Arbeitsbereich der Form an.

Dies sind natürlich alles nur einfache Beispiele zur Anbindung eines BusinessObjectManagers an eine Oberfläche. Im wirklichen Leben muss noch einiges an Drumherum passieren, um dem Anwender eine benutzerfreundliche und stabile Anwendung an die Hand zu geben. Am generellen Prinzip der 3-Schichten Trennung ändert sich jedoch nichts wesentliches mehr. Das vorgestellte Design bietet die Möglichkeit, Oberfläche, Verarbeitung und Datenhaltung strikt zu trennen. Damit können die Verarbeitungsklassen bei einem Wechsel der Oberfläche oder der Datenbank weiterverwendet werden und die Algorithmen der Verarbeitungsschicht können komplett ohne Oberfläche getestet werden.

```
...
private AccountManager _AccountMan = new AccountManager();
...

private void AktualisierenButton_Click(object sender, System.EventArgs
e)
{
   dsAccounts.Clear();
   DataSet _Accounts = _AccountMan.getDataSet("Select * from
Accounts","Accounts");
```

```
    dsAccounts.Merge( Accounts);
}

private void newButton_Click(object sender, System.EventArgs e)
{
    _Account = (Account)_AccountMan.newBusinessObject();
    displayBO();
}

private void saveButton_Click(object sender, System.EventArgs e)
{
    _Account.Data["Customer"] = CustomerTextBox.Text;
    _Account.Data["Project"] = ProjectTextBox.Text;
    _Account.Data["Account"] = AccountTextBox.Text;
    _AccountMan.saveBusinessObject(_Account);

}

private void AccountsGrid_CurrentCellChanged(object sender,
System.EventArgs e)
{
    string _OID =
dsAccounts.Tables[0].Rows[AccountsGrid.CurrentRowIndex]["OID"].ToString(
);
    loadBO(_OID);
    displayBO();
}

private void loadBO(string OID)
{
    _Account = (Account)_AccountMan.getBusinessObject(OID);
}

private void displayBO()
{
    CustomerTextBox.Text = _Account.Data["Customer"].ToString();
    ProjectTextBox.Text = _Account.Data["Project"].ToString();
    AccountTextBox.Text = _Account.Data["Account"].ToString();
}
```

**Listing 6: Methoden des Beispielformulars**

## Zusammenfassung

In ersten Teil meines Artikels über Geschäftsobjekte und deren Realisierung mit .NET habe ich zunächst gezeigt, welche Vorteile eine Architektur mit Geschäftsobjekten bietet. Anschließend habe ich ein Designmodell vorgestellt, das die Verarbeitung konsequent vom Datenzugriff trennt und die Möglichkeit zu einer dynamischen Überprüfung von Geschäftsregeln anbietet. Im zweiten Teil wird das hier vorgestellte Designmodell mit den Klassen von .NET in C# realisiert. Als Abschluss stelle ich eine Beispielapplikation vor, die den Einsatz von Geschäftsobjekten zusammen mit Windows Forms erläutert.

## Literaturhinweise


[BlahaPremerlani98]

Blaha, M. / Premerlani, W.: Object-oriented Modeling and Design for Database Applications; Prentice Hall, 1998

[Gamma95]

Gamma, E. et.al.: Design Patterns, Elements of Reusable Object-Oriented Software; Addison-Wesley, 1995

[McNeish02]

McNeish, K.: .NET for Visual FoxPro Developers; Hentzenwerke Publishing, 2002

[Rätzmann2000]

Rätzmann, M.: Klassen zu Relationen; http://www.raetzmann-gmbh.de/Klassen%20zu%20Relationen.htm