Acceso a Datos
ADO.NET

Diseño e implementación de aplicaciones Web con .NET

Indice

Evolución histórica de acceso a datos

- Inicialmente, no había interfaces comunes de acceso a datos
  - Cada proveedor proporcionaba un API (u otros mecanismos)
  - Cambiar SGBD tenía un coste muy alto

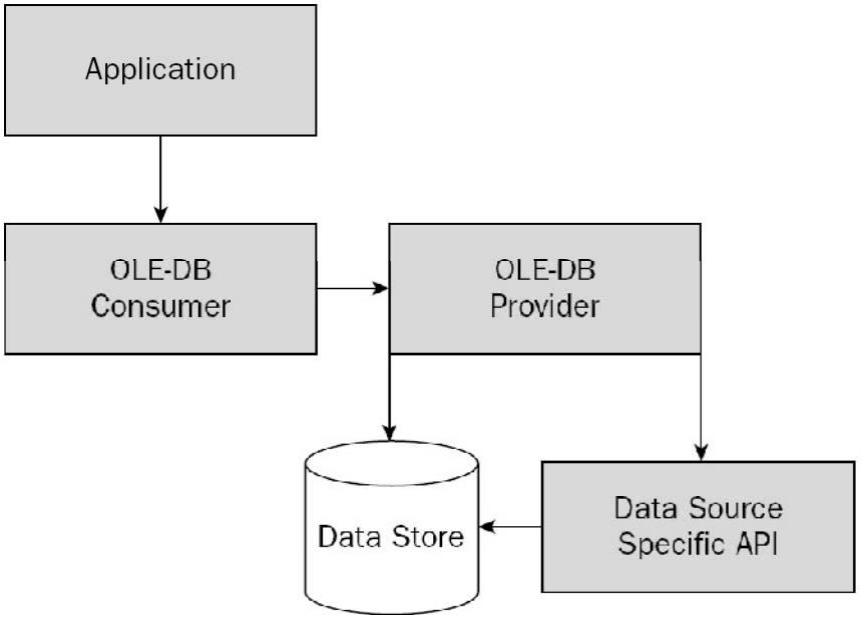
Evolución histórica de acceso a datos
ODBC (Open DataBase Connectivity)

- Estándar de acceso a BD desarrollado por Microsoft
- Proporciona interfaz única para acceder a varios SGBD
  - Modelo de drivers para acceder datos
  - Cualquier proveedor puede escribir un driver ODBC
  - ⇒ Es posible escribir aplicación independiente del SGBD
- Soportado por la mayoría de los SGBD
- No incluye soporte para algunos tipos incluidos en SQL:1999 y SQL:2003

Evolución histórica de acceso a datos
OLE-DB (Object Linking and Embedding Database)

- Desarrollado por Microsoft para mejorar ODBC
- Proporciona un API más limpia y acceso a datos más eficiente que ODBC
- OLE-DB Providers
  - Proporcionan acceso a un SGBD
  - Inicialmente: ODBC
  - Posteriormente se añadiron otros
- OLE-DB Consumers
  - Se comunican con los “proveedores”

Evolución histórica de acceso a datos
OLE-DB (Object Linking and Embedding Database)



Arquitectura OLE-DB

Evolución histórica de acceso a datos
Data Access Consumers

- Desde lenguajes que utilizan punteros, como C o C++ es posible acceder directamente a las APIs ODBC y OLE-DB
- Para acceder desde otros lenguajes es necesaria una nueva capa
  - ⇒ Aparecen: DAO, RDO, ADO y ADO.NET

Evolución histórica de acceso a datos
Data Access Consumers

- Data Access Objects (DAO)
  - Estaba basado en el motor JET, que había sido diseñado para Access
  - DAO 1.0 soportaba ODBC y comunicación directa con Access (sin ODBC)
  - DAO 2.0 se amplió para soportar OLE-DB
  - Problema: sólo puede hablar con el motor JET
    - ⇒ Menor rendimiento

Evolución histórica de acceso a datos
Data Access Consumers

- Remote Data Objects (RDO)
  - Solución de Microsoft al bajo rendimiento de DAO
  - Para comunicarse con BD distintas de Access, RDO no usa el motor JET como DAO. Se comunica con el nivel ODBC directamente
  - Puede usar cursores del lado del cliente para navegar los registros, en contraposición a la necesidad de DAO de usar cursores del lado de servidor

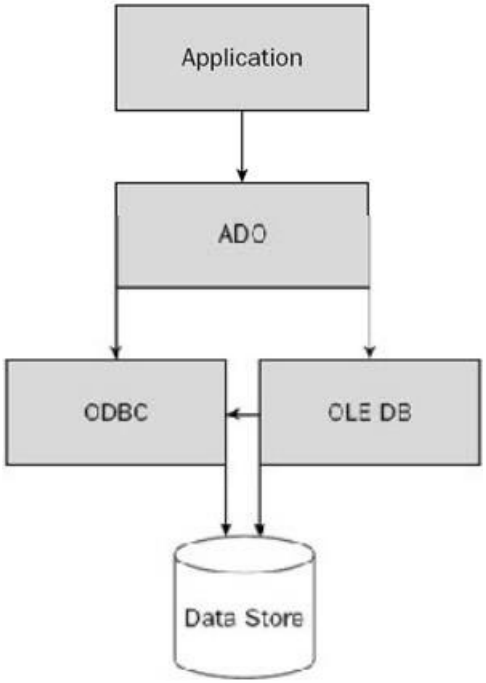
    ⇒ Mejor rendimiento

Evolución histórica de acceso a datos
Data Access Consumers

- ActiveX Data Objects (ADO)
  - Propuesto como sustituto de DAO y RDO
  - Se pretendía que sirviese para acceder a cualquier tipo de datos (desde BD a e-mail, ficheros de texto plano y hojas de cálculo)
  - Soporta comunicación con fuentes de datos a través de ODBC y OLE-DB
  - Introdujo el modelo de proveedores (**provider model**), que permitió a los vendedores de software crear sus propios proveedores

Evolución histórica de acceso a datos
Data Access Consumers

- ActiveX Data Objects (ADO)

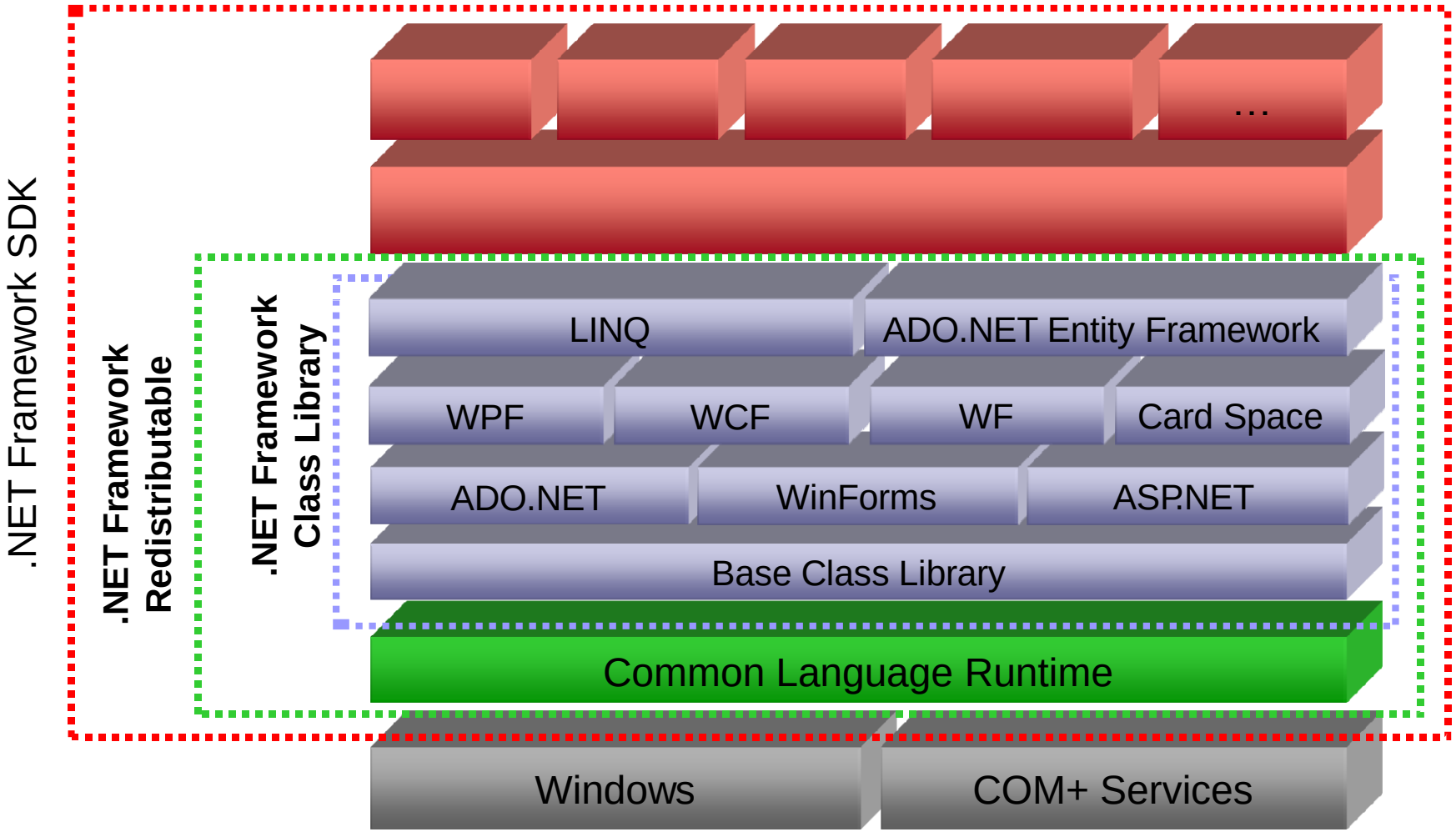


Arquitectura ADO

Evolución histórica de acceso a datos
Data Access Consumers

- ActiveX Data Objects (ADO)
  - Características
    - Modelo más limpio que sus predecesores
    - Batch updating
    - Disconnected Data Access
    - Multiple Recordsets
  - Inconvenientes
    - El trabajo en modo desconectado era engorroso
    - No tenía pool de conexiones
    - Diseño no correctamente factorizado

Conceptos básicos de ADO.NET
Arquitectura del .NET Framework

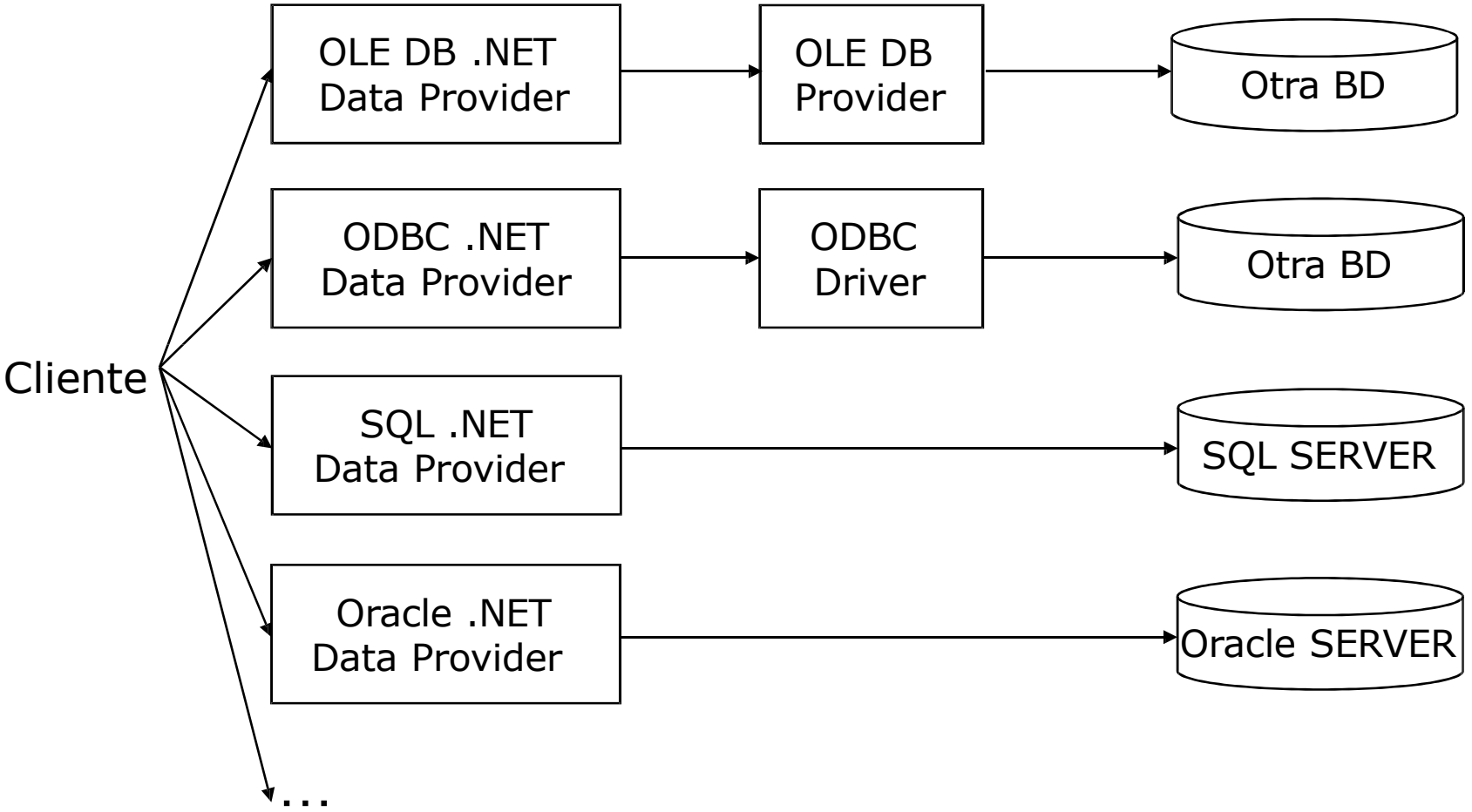


Conceptos básicos de ADO.NET

- Con la versión del .NET Framework, Microsoft introdujo un nuevo modelo de acceso a datos, llamado ADO.NET
  - ADO.NET **no** es ActiveX
    - Es un modelo completamente nuevo (comparte funcionalidad pero no la jerarquía de clases)
  - Soporta comunicación con fuentes de datos a través de ODBC y OLE-DB
  - Además, ofrece la opción de usar proveedores de datos específicos de un SGBD
    - Gran rendimiento al ser capaces de utilizar optimizaciones específicas

  ⇒ Permite conectarse a casi cualquier BD existente

Conceptos básicos de ADO.NET
Data Providers



Proveedores de acceso a datos (data providers) en ADO.NET

Conceptos básicos de ADO.NET
Data Providers

- Proveedores de acceso a datos en ADO.NET
  - OLE DB
    - Acceso vía protocolo OLE DB a cualquier fuente de datos que lo soporte
    - **`System.Data.OleDb`**
  - ODBC
    - Acceso vía protocolo ODBC a cualquier fuente de datos que lo soporte
    - **`System.Data.Odbc`**
  - SQL Server
    - Acceso nativo a MS SQL Server 7.0 o superior y MS Access
    - **`System.Data.SqlClient`**
  - Oracle
    - Acceso nativo a Oracle Server
    - **`System.Data.OracleClient`**
  - Otros provistos por terceros
    - MySQL, PostgreSQL, DB2, etc.

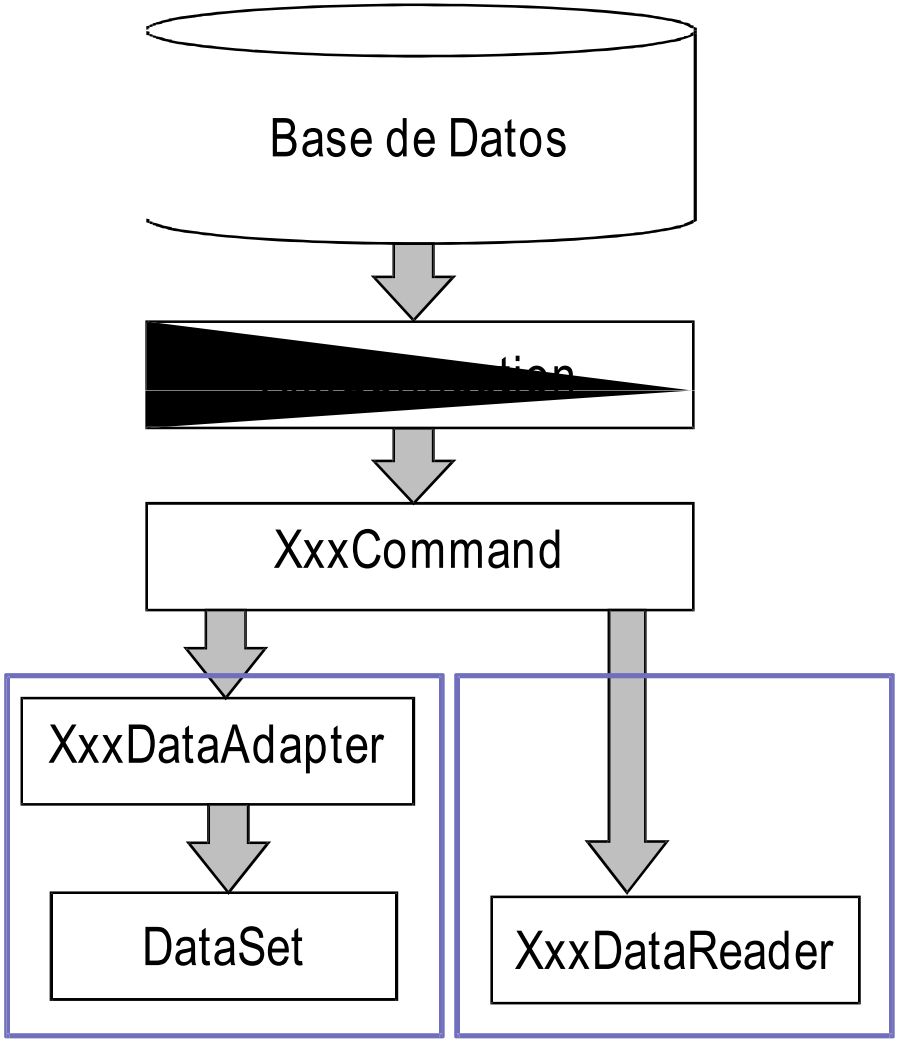
Conceptos básicos de ADO.NET

- Novedades
  - Acceso a datos desconectado real
  - Pool de conexiones

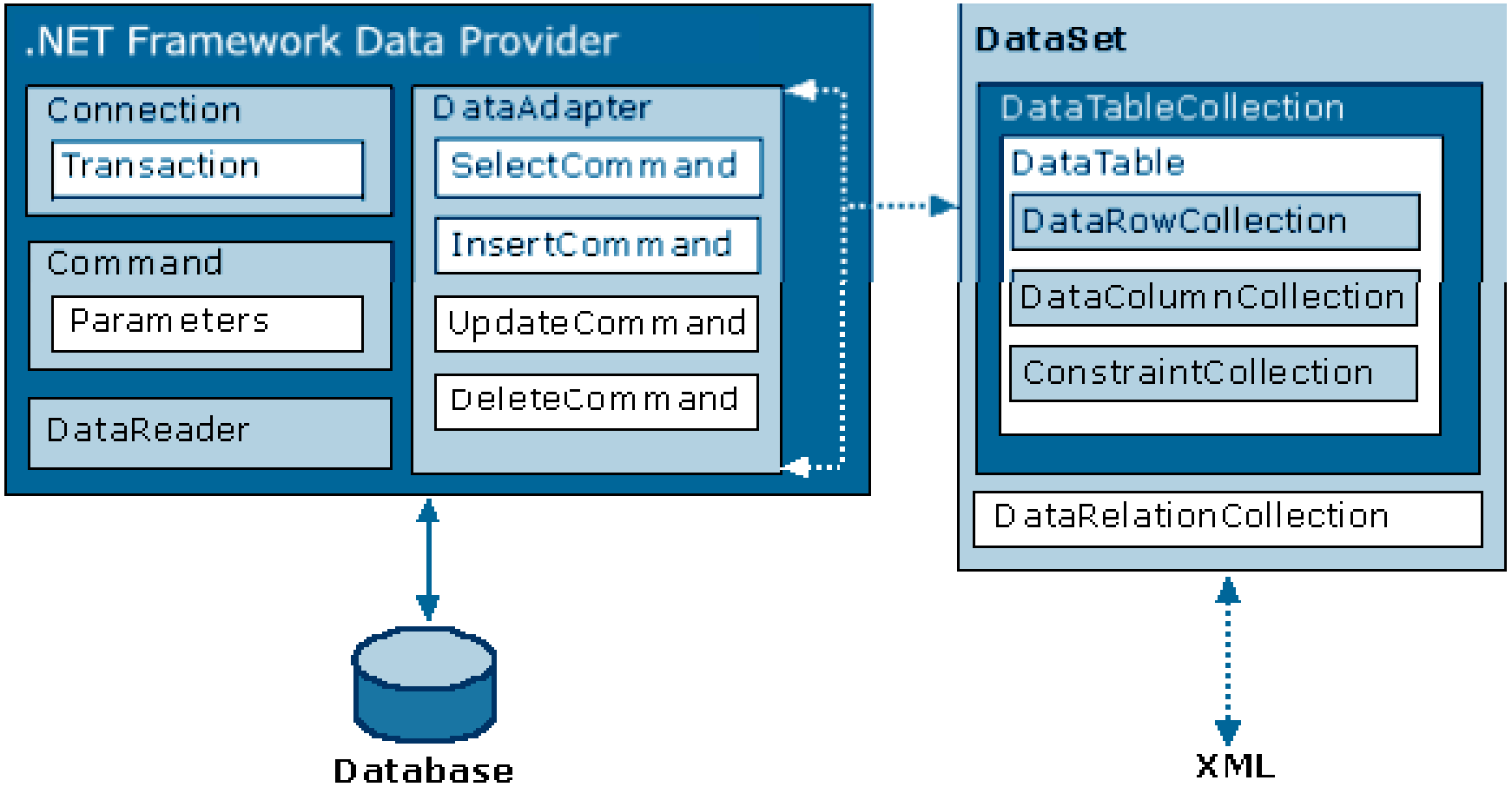
Conceptos básicos de ADO.NET

- **Connection**: responsable de establecer y mantener la conexión a la fuente de datos, junto con cualquier información específica de la conexión
- **Command**: almacena la consulta que va a ser enviada a la fuente de datos y cualquier parámetro aplicable
- **DataReader**: proporciona capacidad de lectura rápida, hacia adelante (*forward-only*) para iterar sobre los registros rápidamente
- **DataSet**: proporciona mecanismo de almacenamiento para datos desconectados.
  - Nunca se comunica con ninguna fuente de datos e ignora la fuente de los datos usada para rellenarlo (*populate*)
- **DataAdapter**: es lo que relaciona el DataSet y la fuente de datos. Es responsable de:
  - Recuperar los datos desde el objeto Command y rellenar el DataSet con los datos recuperados
  - Persistir los cambios realizados en el DataSet en la fuente de datos.

Conceptos básicos de ADO.NET



Conceptos básicos de ADO.NET



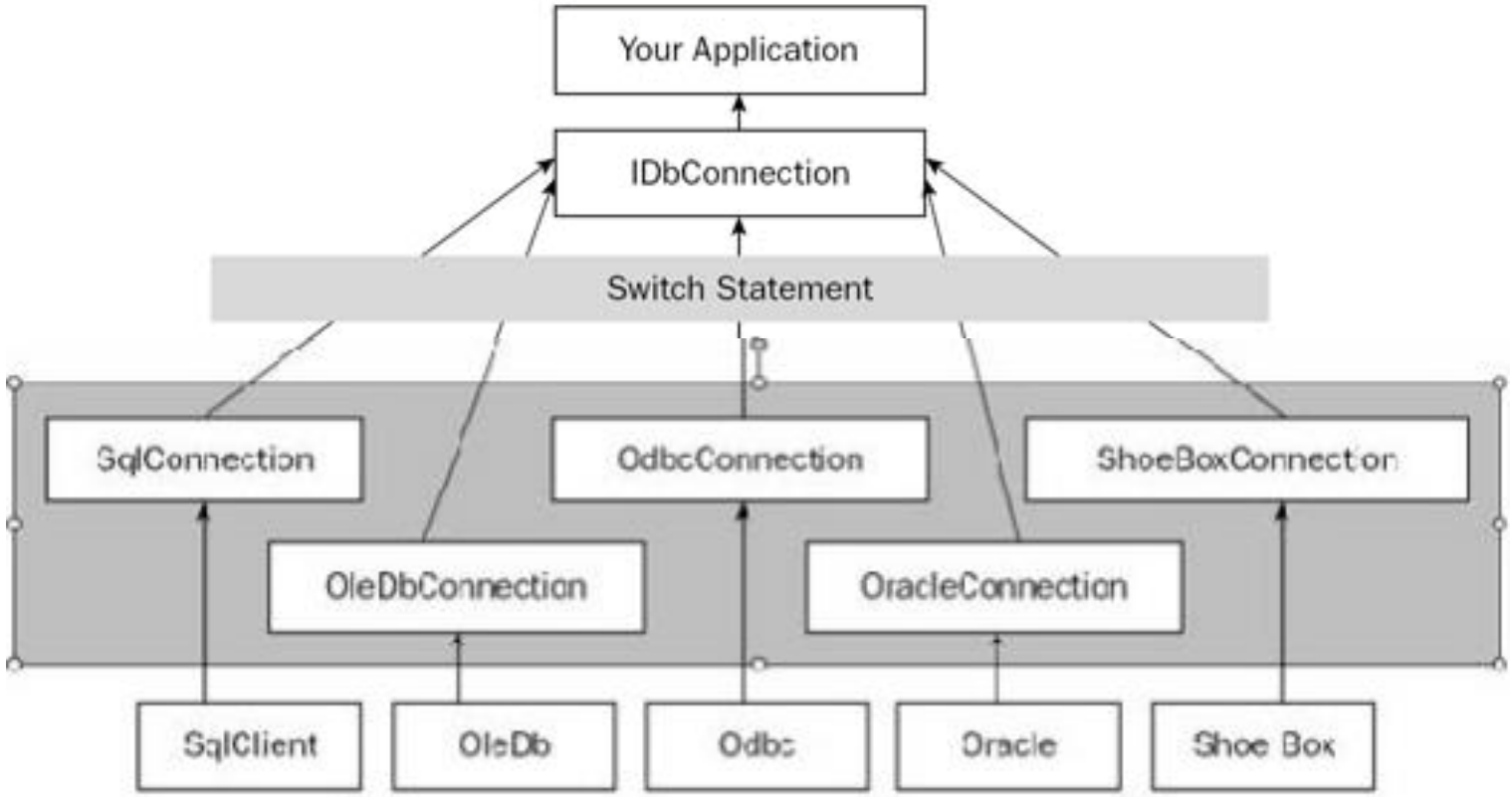
Conceptos básicos de ADO.NET
Entornos de Acceso a Datos

- Conectado:
  - forward-only
  - Aplicación realiza una consulta y lee los datos conforme los va procesando
  - Cursor unidireccional
  - Objeto **`DataReader`**
- Desconectado
  - La aplicación ejecuta la consulta y almacena los resultados de la misma para procesarlos después
  - Minimiza el tiempo de conexión a la base de datos
  - Objetos *lightweight*
  - Objetos **`DataSet`** y **`DataAdapter`**

Conceptos básicos de ADO.NET
Namespace **`System.Data`**

- Organiza el modelo de objetos
- Incluye:
  - **`System.Data`**
  - **`System.Data.OleDb`**
  - **`System.Data.Odbc`**
  - **`System.Data.SqlClient`**
  - **`...`**

Conceptos básicos de ADO.NET
Namespace **`System.Data`**



Independencia del proveedor sin Generic Factory Model
Extraído de: McClure, W. B. (2005). *Professional ADO. NET 2: Programming with SQL Server 2005, Oracle, and MySQL: Wrox.*

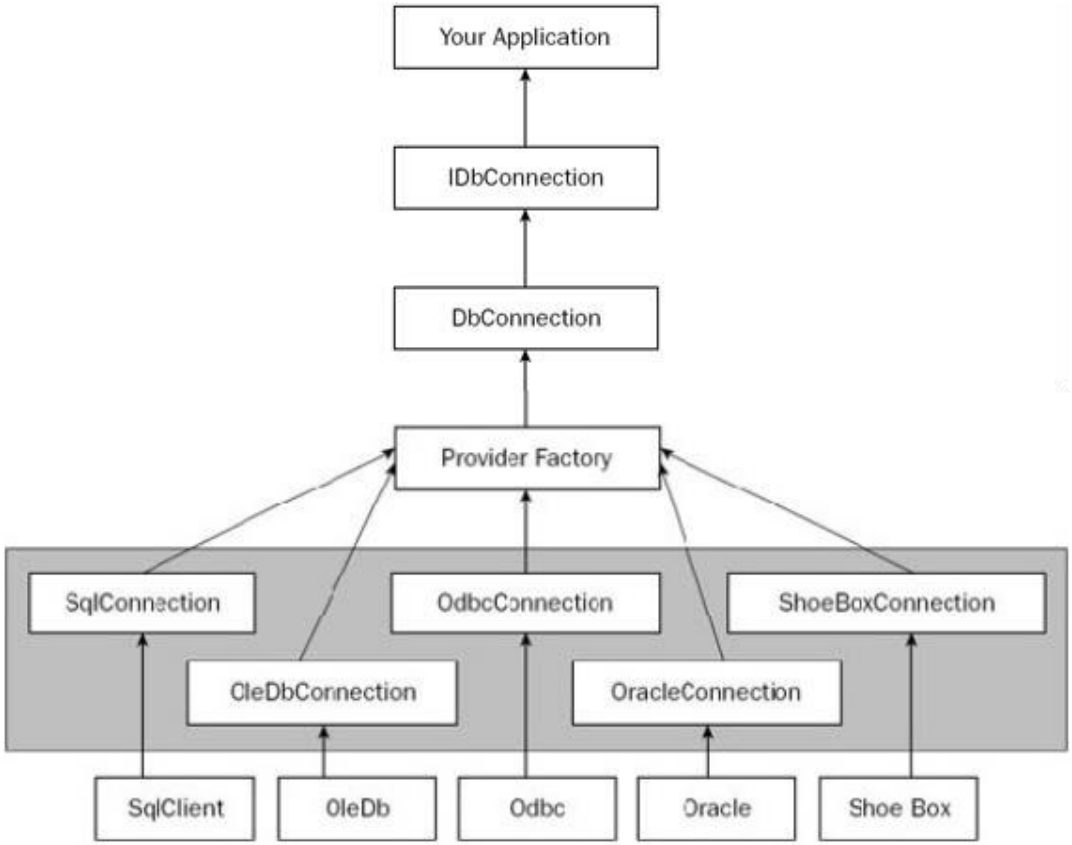
Conceptos básicos de ADO.NET
Namespace **`System.Data`**

- Todos los Data Providers deben implementar una serie de interfaces
  - **`System.Data.IDbConnection`**
  - **`System.Data.IDbCommand`**
  - **`System.Data.IDbDataParameter`**
  - **`System.Data.IDbTransaction`**
  - **`System.Data.IDataReader`**
  - **`System.Data.IDataAdapter`**
  - `...`

ADO.NET 2.0

- 100% compatible con cualquier código escrito en ADO.NET 1.0 ;-)
- Mejora la serialización XML y el pool de conexiones
- **`insert`** permite añadir varias filas en una única instrucción
- Reducción de código necesario para efectuar tareas comunes
- Posibilidad de escribir código de acceso a BD independiente del proveedor
  - ⇒ **Generic Factory Model**

Generic Factory Model



Independencia del proveedor con Generic Factory Model
Extraído de: McClure, W. B. (2005). *Professional ADO. NET 2: Programming with SQL Server 2005, Oracle, and MySQL: Wrox.*

Generic Factory Model

- Namespace **`System.Data.Common`**
  - **`DbConnection`**
  - **`DbCommand`**
  - **`DbParameter`**
  - **`DbDataReader`**
  - **`DbDataAdapter`**
  - **`DbProviderFactories`**
  - **`DbProviderFactory`**
  - **`DbException`**
  - `...`

DbConnection

- Establece una sesión con una fuente de datos
- Implementada por **`SqlConnection`**,**`OdbcConnection`**, **`OleDbConnection`**, etc.
- Funcionalidad
  - Abrir y Cerrar conexiones
  - Gestionar Transacciones

DbConnection

- Propiedades
  - **ConnectionString**
  - **ConnectionTimeOut**
  - **DataBase**
  - **State**
    - **Open**
    - **Close**
- Métodos
  - **void Open()**
  - **void Close()**
  - **void ChangeDataBase(dbName);**
  - **DbCommand CreateCommand()**

DbConnection

Connection String

- Dependerá del proveedor de acceso a datos
- Listas de *conection string's* disponibles en:
  - http://www.codeproject.com/KB/database/connectionstrings.aspx
  - http://www.carlprothman.net/Default.aspx?tabid=81
- Ejemplos:
  - SqlServer:

```
"Data Source=localhost\SQLExpress; Initial Catalog=miniportal;  User
 ID=user;Password=password"

"Data Source=localhost\SQLExpress; Initial Catalog=miniportal;
 Integrated Security=true"
```

  - MySQL ODBC Driver:

```
"DRIVER={MySQL ODBC 3.51 Driver}; SERVER=localhost; PORT=3306;
 UID=user; PWD=password; DATABASE=db;"
```

  - Access OLEDB:

```
"Provider=MSDASQL; Driver={Microsoft Access Driver (*.mdb)};
 Dbq=drive:\path\file.mdb; Uid=user; Pwd=password";
```

DbConnection

Connection Pooling

- Pool de Conexiones **habilitado automáticamente**
  - Pool se crea en base a la cadena de conexión. Ejemplo:

```
DbConnection northwindConnection = new
SqlConnection(); northwindConnection.ConnectionString
=
    "Integrated Security=SSPI;Initial Catalog=northwind";
// Pool A is created.
northwindConnection.Open();

DbConnection pubsConnection = new
SqlConnection(); pubsConnection.ConnectionString
=
    "Integrated Security=SSPI;Initial Catalog=pubs";
// Pool B is created because the connection strings differ.
pubsConnection.Open();

DbConnection otherNorthwindConnection = new
SqlConnection(); otherNorthwindConnection.ConnectionString
=
    "Integrated Security=SSPI;Initial Catalog=northwind";
// The connection string matches pool A.
otherNorthwindConnection.Open();
```

  - Al cerrar una conexión, ésta se devuelve al pool

Proveedores de Acceso a Datos (*Data Providers*)

- Independizar código del proveedor de datos
  - Factoría de proveedores : **`DbProviderFactories`**
  - Crea instancias de un proveedor de acceso a datos

```
/* Returns an instance of a
System.Data.Common.DbProviderFactory
 * for the specified providerName
 */
DbProviderFactory dbFactory =
    DbProviderFactories.GetFactory(providerName);
```

- Objeto **`DbProviderFactory`**
  - **`.CreateCommand()`**
  - **`.CreateConnection()`**
  - **`.CreateParameter()`**

Proveedores de Acceso a Datos (*Data Providers*)

```
public static void Main(String[] args)
{
    DbConnection connection = null;

    try
    {
        /* The providerName is the invariant name of a provider
         * It could be obtained from a configuration file ...
         */
        String providerName = "System.Data.SqlClient";

        // The connection string should be read from a configuration file...
        String connectionString = "Data Source=localhost\\SQLExpress;" +
            "Initial Catalog=test;User ID=testUser;Password=password";

        /* Returns an instance of a System.Data.Common.DbProviderFactory
         * for the specified providerName
         */
        DbProviderFactory dbFactory =
            DbProviderFactories.GetFactory(providerName);

        // Create the connection ...
        connection = dbFactory.CreateConnection();
        connection.ConnectionString = connectionString;
```

Proveedores de Acceso a Datos (*Data Providers*)

```
            // Create the command and set properties ...

            // Open the connection ...
            connection.Open();

            // ...

        }
        catch (Exception e)
        {
            // ...
        }
        finally
        {
            // Ensures connection is closed
            if (connection != null) connection.Close();
        }

}
```

Proveedores de Acceso a Datos (*Data Providers*)

```
public static void Main(string[] args)
{
    try
    {
        DataTable factoryTable = DbProviderFactories.GetFactoryClasses();

        // Lists DataTable information...
        foreach (DataRow dr in factoryTable.Rows)
        {
            Console.WriteLine("Name: {0}", dr["Name"]);
            Console.WriteLine("Description: {0}", dr["Description"]);
            Console.WriteLine("InvariantName: {0}", dr["InvariantName"]);
            Console.WriteLine("AssemblyQualifiedName: {0}",
                              dr["AssemblyQualifiedName"]);
            Console.WriteLine("----------------------------------------");
        }
    }
    catch (Exception e)
    {
        Console.WriteLine("Message: " + e.Message);
        Console.WriteLine("StackTrace: " + e.StackTrace);
    }
    Console.ReadLine();
}
```

Proveedores de Acceso a Datos (*Data Providers*)

- Ejemplo de salida

```
Name: Odbc Data Provider
Description: .Net Framework Data Provider for Odbc
InvariantName: System.Data.Odbc
AssemblyQualifiedName: System.Data.Odbc.OdbcFactory, System.Data,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
---------------------------------------
Name: OleDb Data Provider
Description: .Net Framework Data Provider for OleDb
InvariantName: System.Data.OleDb
AssemblyQualifiedName: System.Data.OleDb.OleDbFactory, System.Data,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
---------------------------------------
Name: SqlClient Data Provider
Description: .Net Framework Data Provider for SqlServer
InvariantName: System.Data.SqlClient
AssemblyQualifiedName: System.Data.SqlClient.SqlClientFactory,
System.Data, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089

< ... >
```

Comandos. **`DbCommand`**

- Representa una sentencia que se envía a una fuente de datos
  - Generalmente, pero no necesariamente SQL
- Implementada por **`SqlCommand`**, **`OleDbCommand`**, etc.
- Funcionalidad
  - Definir la sentencia a ejecutar
  - Ejecutar la sentencia
  - Enviar y recibir parámetros
  - Crear una versión compilada

Comandos. **DbCommand**

- Propiedades
  - **CommandText**
  - **CommandTimeOut**
  - **CommandType**
    - **CommandType.Text**
    - **CommandType.StoredProc**
  - **Connection**
  - **Parameters**
  - **Transaction**

Comandos. **DbCommand**

- Si se trabaja con comandos dependientes del Data Provider es posible disponer de varios constructores
  - **SqlCommand()**
  - **SqlCommand(cmdText)**
    - **e.g:**

```
SqlCommand command = new SqlCommand("SELECT loginName " +
    "FROM UserProfile ", connection);
```

  - **SqlCommand(cmdText, connection)**
  - **SqlCommand(cmdText, connection, transaction)**

Comandos. **DbCommand**

- Si se trabaja con comandos genéricos (independientes del *Data Provider*), el comando debe crearse a partir de la conexión
  - Único constructor, sin parámetros
  - Inicialización mediante acceso a propiedades

```
// Create the command and set properties ...
DbCommand command = connection.CreateCommand();

command.CommandText = "SELECT loginName FROM UserProfile ";

command.Connection = connection;

command.CommandTimeout = 15;

command.CommandType = CommandType.Text;

// Open the connection ...
connection.Open();
```

Comandos

`Command.Prepare()`

- Debería usarse cuando un comando se ejecuta múltiples veces
- Origina una sobrecarga inicial debida a la creación de un procedimiento almacenado en el SGBD para la ejecución del comando
  - Se rentabiliza en las siguientes ejecuciones del comando
- La ejecución de **`Command.Prepare()`** necesita una conexión abierta y disponible
- Requiere especificar el tamaño de los parámetros empleados en el comando, mediante la propiedad **`DbParameter.Size`**

Parámetros

- Comandos poseen colección **`Parameters`**
- **`DbParameter`**
  - **`ParameterName`**
  - **`DbType`**
    - Enumeración: **`String`**, **`Int32`**, **`Date`**, **`Double`**, etc.
  - **`Value`**
  - **`Size`**
  - **`IsNullable`**

Parámetros

```
// Create the command and set properties ...
DbCommand command = connection.CreateCommand();
command.CommandText =
    "SELECT loginName FROM UserProfile " +
    "WHERE loginName = @loginName ";

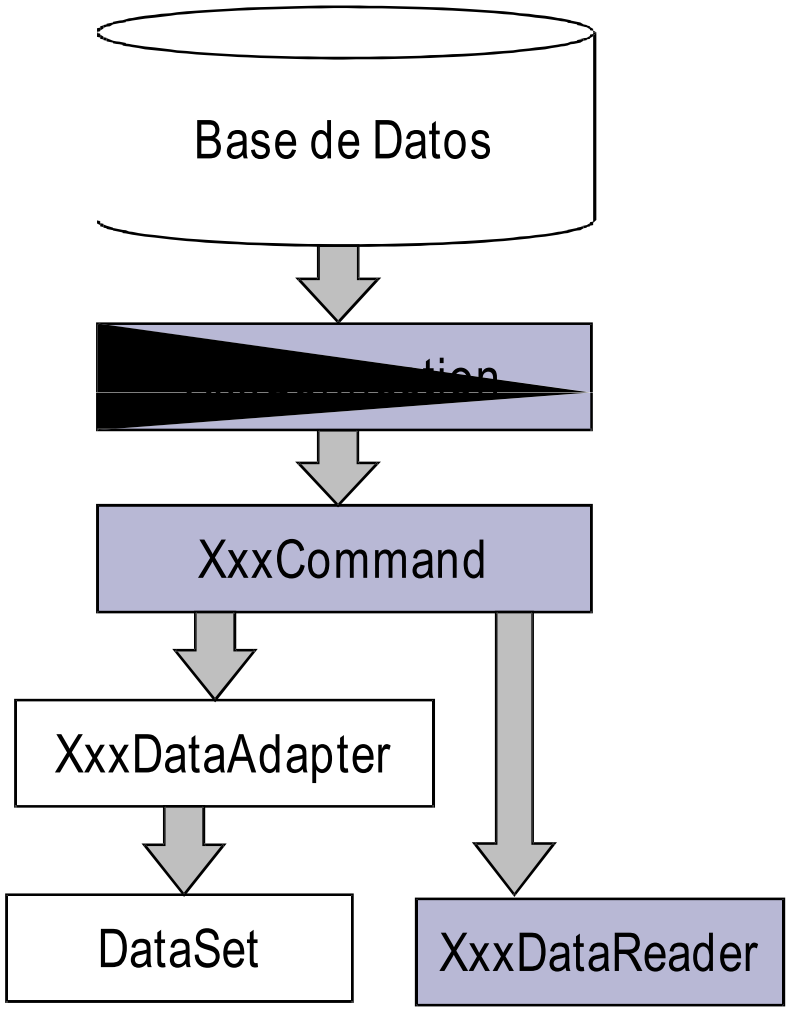
// Create and populate parameter
DbParameter loginNameParam = command.CreateParameter();
loginNameParam.ParameterName = "@loginName";
loginNameParam.DbType = DbType.String;
loginNameParam.Value = "loginTest";
/* If command.Prepare() is used then paramSize must be greater
 * than 0
 */
loginNameParam.Size = 30;

command.Parameters.Add(loginNameParam);

// Open the connection ...
connection.Open();

/* Prepare the command to have better performance. The provider
 * will use the prepared (compiled) version of the command for
 * any subsequent executions.
 * Notice that Prepare() requires and open and available
 * connection.
 */
command.Prepare();
```

Entorno Conectado



- **XxxConnection:** maneja la conexión a una BD
- **XxxCommand**: ejecuta comandos contra una BD
- **XxxDataReader**: Proporciona acceso a datos Read-only, Forward-only (Entorno Conectado)

Entorno Conectado. **`DbCommand`**

- **`ExecuteReader`**
  - Sentencias que devuelven múltiples filas de resultados (**`DbDataReader`**)
- **`ExecuteNonQuery`**
  - Sentencias UPDATE, DELETE, etc. que no devuelven ninguna fila como resultado
- **`ExecuteScalar`**
  - Sentencias SQL que devuelven una fila con un único valor como resultado

Entorno Conectado. **`DbDataReader`**

- Proporciona acceso secuencial de sólo lectura a una fuente de datos
- Creado a través de **`command.ExecuteReader()`**
- Al utilizar un objeto **`DbDataReader`**, las operaciones sobre la conexión **`DbConnection`** quedan deshabilitadas hasta que se cierre el objeto **`DbDataReader`**

Entorno Conectado. **`DbDataReader`**

- Propiedades de interés:
  - **`FieldCount:`** devuelve el número de campos en la fila actual
  - **`RecordsAffected`**: número de registros afectados
- Métodos
  - **`Read()`**
    - Avanza el **`DbDataReader`** al siguiente registro
    - Inicialmente se sitúa antes del primer registro del resultado
    - Devuelve **`false`** si ha llegado al final, **`true`** en caso contrario
  - **`Close`**()
    - Cierra el objecto **`DbDataReader`**
  - **`GetValues`**()
    - Obtiene la fila actual
  - Proporciona métodos para el tipado de los datos leídos (**`GetValue`**, **`GetString`**, etc.)
    - **`Boolean b = myDataReader.GetBoolean(fieldNumber);`**

Entorno Conectado. **ExecuteReader()**

```
try
{
    // ...

    // Create the command and set properties ...
    DbCommand command = connection.CreateCommand();
    command.CommandText =
        "SELECT loginName, email " +
        "FROM UserProfile";
    command.Connection = connection;

    // Open the connection and execute the command ...
    connection.Open();
    DbDataReader dr = command.ExecuteReader();

    // Data access
    while (dr.Read())
    {
        String loginName = dr.GetString(0);
        String email = dr.GetString(1);
        Console.WriteLine("loginName: " + loginName +
            ", email: " + email);
    }

    // Close the DataReader ...
    dr.Close();
}
```

Entorno Conectado. **ExecuteNonQuery**

```
try
{
    // ...

    // Create the command and set properties ...
    // SQL Update Command modifies data but it does not return any  data
    DbCommand command = connection.CreateCommand();
    command.CommandText =
        "UPDATE Account " +
        "SET balance = 1.1 * balance ";

    command.Connection = connection;

    // Open the connection and execute the command ...
    connection.Open();

    /* Executes a SQL statement against the Connection object
     * of a .NET Framework data provider, and returns the number
     * of rows affected.
     */
    int affectedRows = command.ExecuteNonQuery();

    Console.WriteLine("affectedRows: " + affectedRows);
}
```

Entorno Conectado. **ExecuteScalar**

```
try
{
    // ...

    // Create the command and set properties ...
    DbCommand command = connection.CreateCommand();
    command.CommandText =
        "SELECT count(*) " +
        "FROM UserProfile ";

    command.Connection = connection;

    // Open the connection and execute the  command ...
    connection.Open();

    /* Executes the query, and returns the first column of the
     * first row in the resultset returned by the query (as an
     * object).
     * Additional columns or rows are ignored.
     */
    int numberOfUsers = (int)command.ExecuteScalar();

    Console.WriteLine("numberOfUsers: " + numberOfUsers);
}
```

Transacciones

- Transacción:
  - Conjunto sentencias que constituyen una unidad lógica de trabajo
- Deben cumplir las propiedades ACID:
  - **A**tomicity:
    - Las sentencias se ejecutan todas o ninguna
  - **C**onsistency:
    - Una vez finalizada, los datos deben ser consistentes
  - **I**solation:
    - Transacciones se comportan como si cada una fuera la única transacción
  - **D**urability:
    - Una vez finalizada, los cambios son permanentes

Transacciones

- Se crean a partir de la conexión
  - **`connection.BeginTransaction();`**
- Es obligatorio asociar los comandos a la transacción
  - Propiedad **`command.Transaction`**
- Métodos
  - **`Commit();`**
  - **`Rollback();`**

Transacciones

- Niveles de Aislamiento
  - **`IsolationLevel.ReadUncommitted`**: pueden ocurrir “dirty reads”, “non-repeatable reads” y “phantom reads”
  - **`IsolationLevel.ReadCommitted`**: pueden ocurrir “non-repeatable reads” y “phantom reads”
  - **`IsolationLevel.RepeatableRead`**: pueden ocurrir “phantom reads"
  - **`IsolationLevel.Serializable`**: elimina todos los problemas de concurrencia
- El nivel de aislamiento se fija en el momento de crear la transacción
  - ```
    connection.BeginTransaction(
    IsolationLevel.Serializable);
    ```
    ```

Transacciones

```
try
{
    // ...

    // Open the connection ...
    connection.Open();

    // Starts a new transaction ...
    // transaction = connection.BeginTransaction();      //default
    transaction = connection.
        BeginTransaction(IsolationLevel.Serializable);

    // Create the command and set properties ...
    DbCommand selectCommand = connection.CreateCommand();
    selectCommand.Connection = connection;
    selectCommand.CommandText =
        "SELECT balance " +
        "FROM ACCOUNT " +
        "WHERE accId = 1";

    // Associate the command with the transaction
    selectCommand.Transaction = transaction;
```

Transacciones

```
// Execute the selectCommand ...
dataReader = selectCommand.ExecuteReader();

if (!dataReader.Read())
{
    throw new Exception("Error in DataBase access!");
}

balance = dataReader.GetDouble(0);
Console.WriteLine("Actual balance: " + balance);
balance = 1.1 * balance;
Console.WriteLine("New balance must be: " + balance);

/* PAUSE: another process should change the balance
 * TIP : use sql server management studio to launch a query
 * which change the balance. Notice that in this case SQL Server
 * performs a row-level block, so you can modify the balance
 * of other account.
 */
Console.ReadLine();
```

Transacciones

```
        // DataReader must be closed before the updateCommand execution
        dataReader.Close();

        // Create the updateCommand and set properties ...
        DbCommand updateCommand = connection.CreateCommand();
        updateCommand.Connection = connection;
        updateCommand.CommandText =
            "UPDATE ACCOUNT " +
            "SET balance = @balance " +
            "WHERE (accId = 1)";
        updateCommand.Transaction = transaction;

        DbParameter balanceParam = updateCommand.CreateParameter();
        balanceParam.ParameterName = "@balance";
        balanceParam.DbType = DbType.Decimal;
        balanceParam.Value = balance;
        updateCommand.Parameters.Add(balanceParam);

        // Execute the updateCommand ...
        int afectedRows = updateCommand.ExecuteNonQuery();

        transaction.Commit();
        commited = true;

        Console.WriteLine("Transaction COMMITED");
    }
```

Transacciones

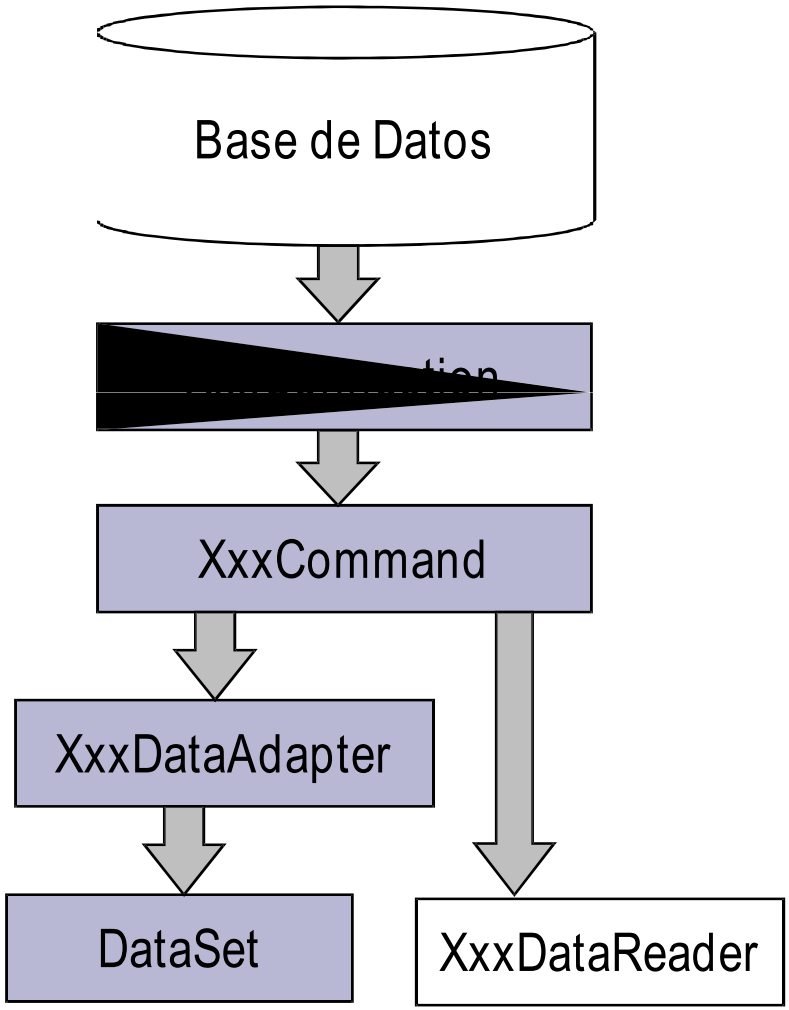
```
catch (DbException e)
{
    Console.WriteLine(e.StackTrace);
    Console.WriteLine(e.Message);
}
catch (Exception e)
{
    Console.WriteLine(e.StackTrace);
    Console.WriteLine(e.Message);
}
finally
{
    if (!commited)
    {
        if (transaction != null)
        {
            transaction.Rollback();
        }
    }

    // Ensures connection is closed
    if (connection != null)
    {
        connection.Close();
    }
}
```

Excepciones

- **`System.Data.Common.DbException`**
  - Se lanza cuando ocurre algún problema en la capa de acceso a datos
  - Es una clase abstracta que implementa **`ExternalException`**
  - Cada "*Data Provider*" proporcionará una implementación específica
- Constructores:
  - **`DbException()`**
  - **`DbException(string message)`**
    - **`message`**: mensaje a mostrar
  - **`DbException(string message, Exception innerException)`**
    - **`innerException`**: la referencia de la excepción interna
  - **`DbException(string message, int errorCode)`**
    - **`errorCode`**: código de error para la excepción

Entorno Desconectado



- **XxxConnection:** maneja la conexión a una BD
- **XxxCommand**: ejecuta comandos contra una BD
- **XxxDataAdapter**: intercambia datos entre un DataSet y una BD
- **DataSet**: copia local de datos relacionales (Entorno Desconectado)

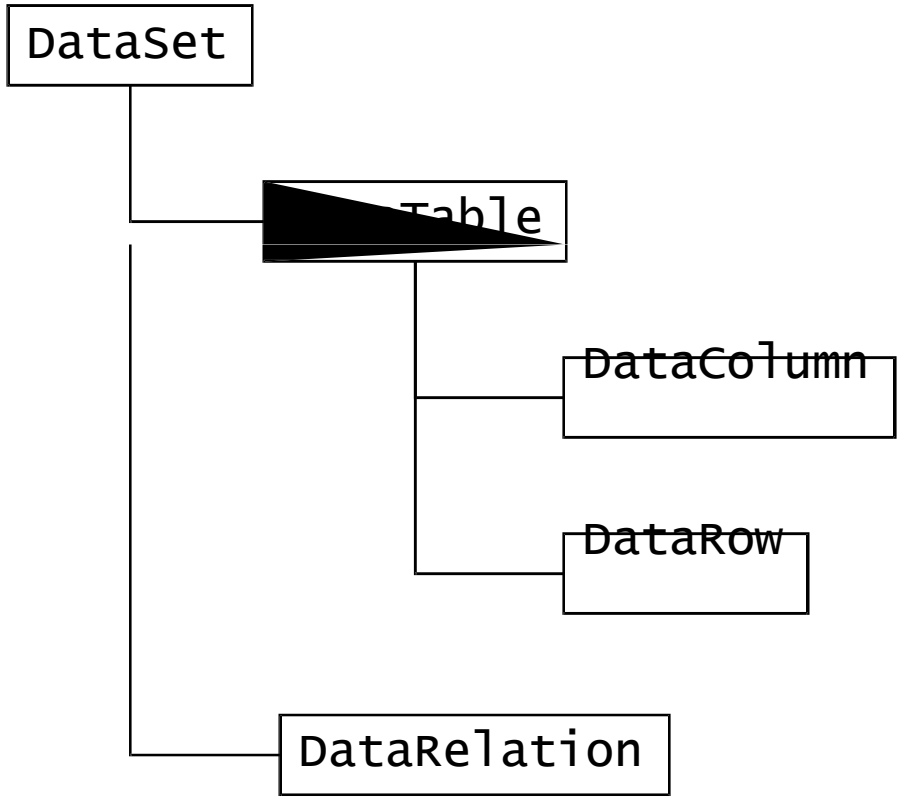
Entorno Desconectado: **`DataSet`**

- Núcleo ADO.NET bajo entorno desconectado
- Representación en memoria del contenido de la base de datos
- Operaciones sobre los datos se realizan sobre el **`DataSet`**, no sobre el origen de datos
- Almacena
  - Tablas (**`DataTable`**)
  - Relaciones ente tablas (**`DataRelation`**)
- Independiente del proveedor de datos
- Problemas
  - Sincronización datos
  - Acceso concurrente

Entorno Desconectado: **`DataTable`**

- Representación lógica de una tabla de la base de datos
- Propiedades de interés:
  - **`Columns:`**
    - Colección de tipo **`ColumnsCollection`** de objetos **`DataColumn`**
  - **`Rows`**:
    - Colección de tipo **`RowsCollection`** de objectos **`DataRow`**
  - **`ParentRelations`**:
    - **`RelationsCollection`**. Relaciones en las que participa la tabla
  - **`Constraints`**:
    - Colección de tipo **`ConstraintsCollection`**
  - **`DataSet`**:
    - **`DataSet`** en el que está incluida la **`DataTable`**
  - **`PrimaryKey`**:
    - **`DataColumn`** que actúa como clave primaria de la tabla

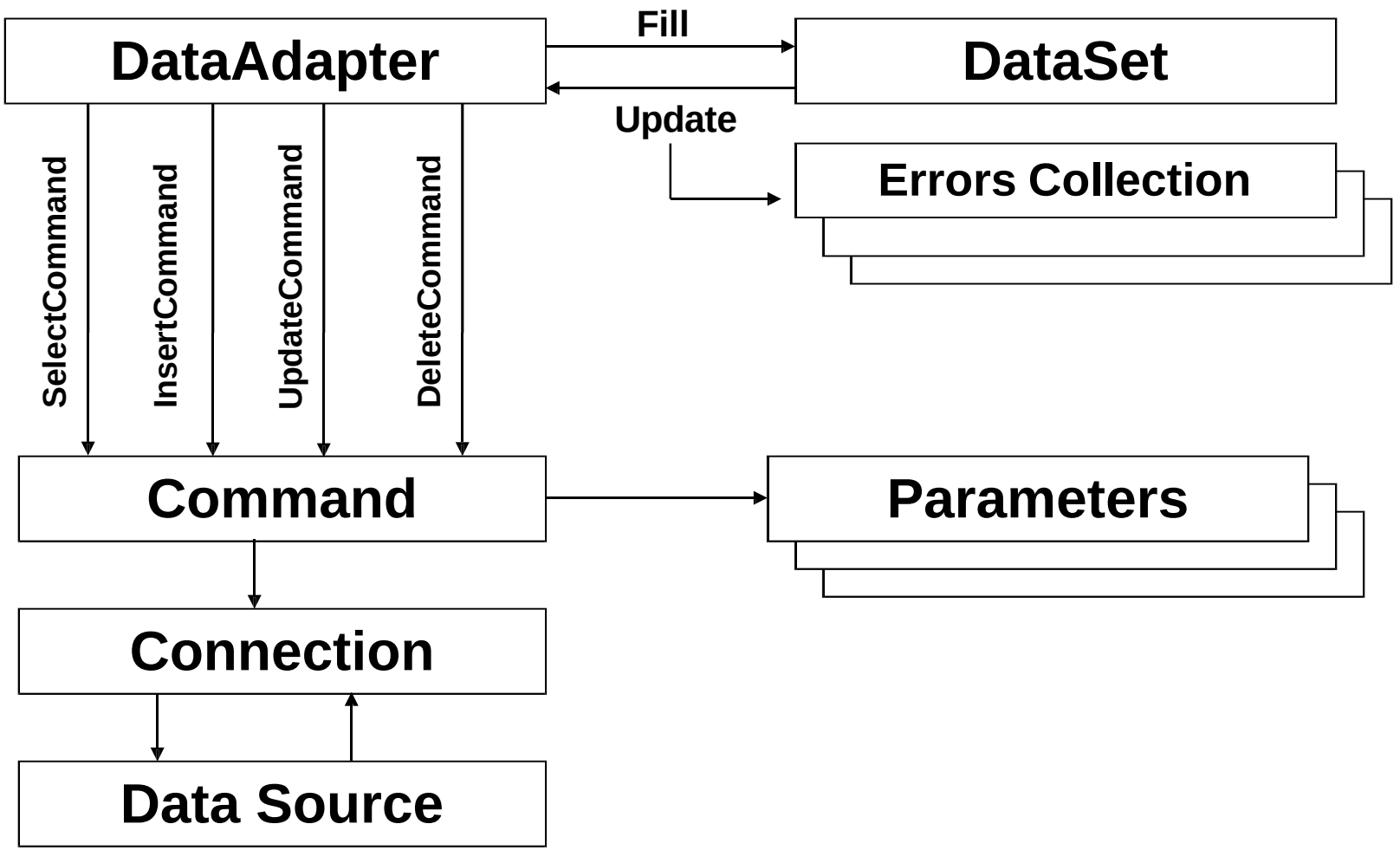
Entorno Desconectado: **DataSet**



Entorno Desconectado. **`XxxDataAdapter`**

- Bridge entre origen de datos y **`DataSet`**
- Implementa los métodos abstractos de la clase **`DataAdapter`**:
  - **`public abstract int Fill( DataSet dataSet );`**
  - **`public abstract int Update( DataSet dataSet );`**
- Propiedades de interés:
  - **`DeleteCommand:`** El comando de borrado, expresado en SQL
  - **`InsertCommand`**: Obtiene o establece el comando de inserción
  - **`SelectCommand`**: Obtiene o establece el comando de selección
  - **`UpdateCommand`**: Obtiene o establece el comando de actualización
  - **`TableMappings`**: Relaciona la tabla con el **`DataTable`**
- Debe especificarse siempre un comando de selección

Entorno Desconectado



Entorno Desconectado

```
try
{
    // ...

    // Create the connection ...
    connection = dbFactory.CreateConnection();
    connection.ConnectionString = connectionString;

    DbCommand selectCommand = connection.CreateCommand();
    selectCommand.CommandText =
        "SELECT * " +
        "FROM UserProfile";

    // Create and configure the  DataAdapter...
    DbDataAdapter dataAdapter  = dbFactory.CreateDataAdapter();
    dataAdapter.SelectCommand  = selectCommand;

    // Create the DataSet (it will store a copy of database values) ...
    DataSet dataSet = new DataSet("UserProfileDS");

    // Fill DataSet
    dataAdapter.Fill(dataSet, "Users");

    // ... now connection with database is automatically closed.
```

Entorno Desconectado

```
        /* Changes are applied only within dataset (database does not
         * change, so the next execution will return the same initial
         * value)
         */
        DataRow firstRow = dataSet.Tables["Users"].Rows[0];
        Console.WriteLine("LoginName read from database: " +
            firstRow["loginName"]);

        firstRow["loginName"] = "newLoginName";

        foreach (DataRow dr in dataSet.Tables["Users"].Rows)
        {
            Console.WriteLine("LoginName changed in DataSet: " +
                dr["loginName"]);
        }

}

// catch ...

// finally ...
```