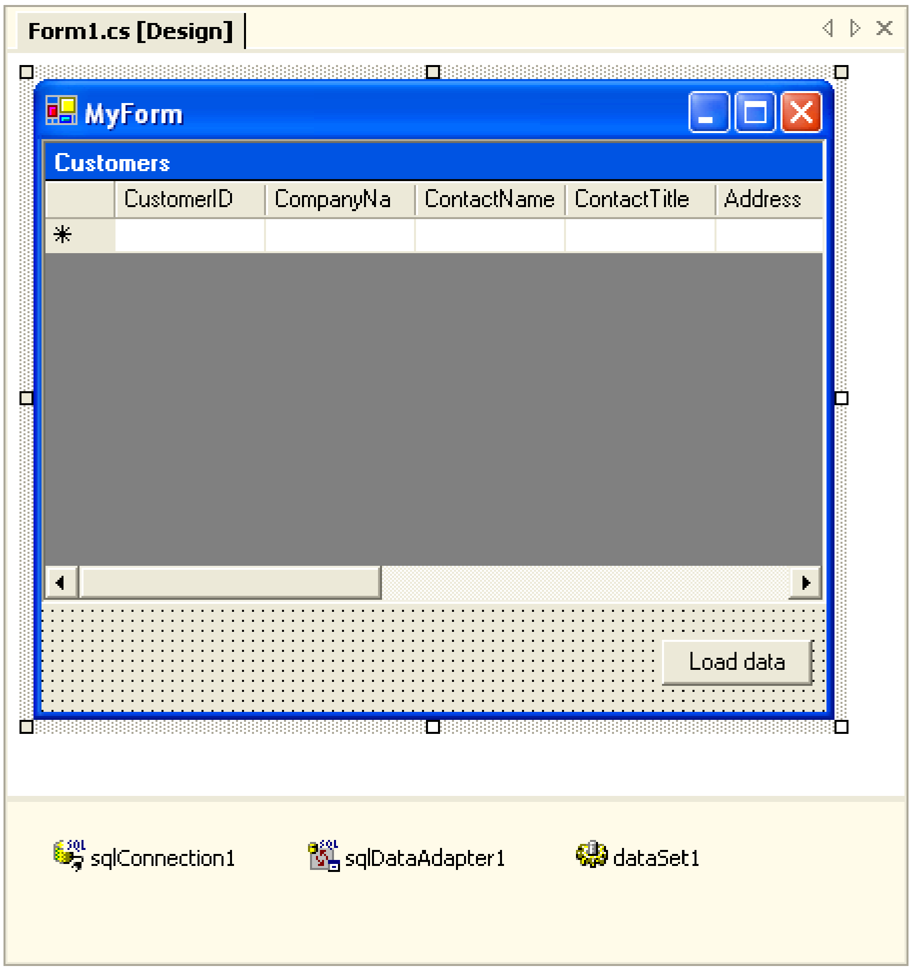
Vanilla DAL Tutorial

The Vanilla Data Access Layer For .NET
http://sourceforge.net/projects/vanilla-dal/

Motivation

Visual Studio .NET contains some impressive rapid application development features for database applications, e.g. dropping a database table from the server toolbox onto a WinForm automatically creates a typed Dataset plus a ready-to-go DataAdapter, including corresponding SQL code. After binding the Dataset to a grid and invoking the DataAdapter's Fill()-method (one line of hand-written code) the application loads and displays database content:



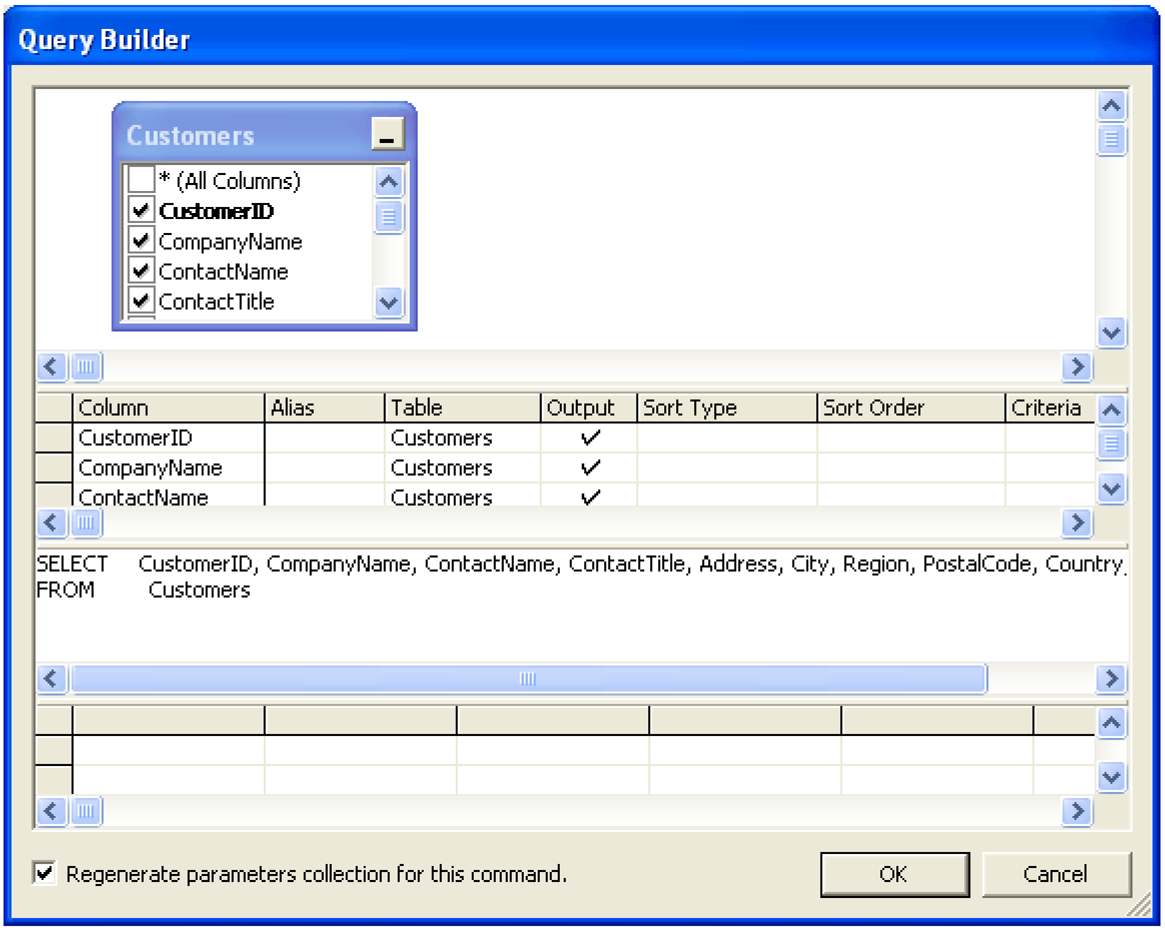
Picture 1: Mixing UI and Data Access Code in Visual Studio .NET

There are also some drawbacks with this approach. For example, it is considered bad design to have SQL code and database connection strings cluttered all over Windows forms (no information hiding, no possibility for re-use). It is better not to drop tables on a form, but on a .NET Component class, and to make the connection string a dynamic property and bind it to a .config-file value.

Visual Studio .NET 2005 has improved as it does not produce SQL code inside WinForms or ASP.NET pages, but manages project-wide datasources, and places SQL code in so called TableAdapters, which are coupled with the underlying typed Dataset (actually the TableAdapter's code resides in a partial Dataset class file). Whether this is the best choice remains questionable.

Another RAD feature is the so called SQL query builder. It’s either used for manually editing

SQL code or for auto-generating select-, insert-, update- and delete-statements based on database schema information. Here we can see query builder in action:



Customers:
- [ ] * (All Columns)
- [✓] **CustomerID**
- [✓] CompanyName
- [✓] ContactName
- [✓] ContactTitle

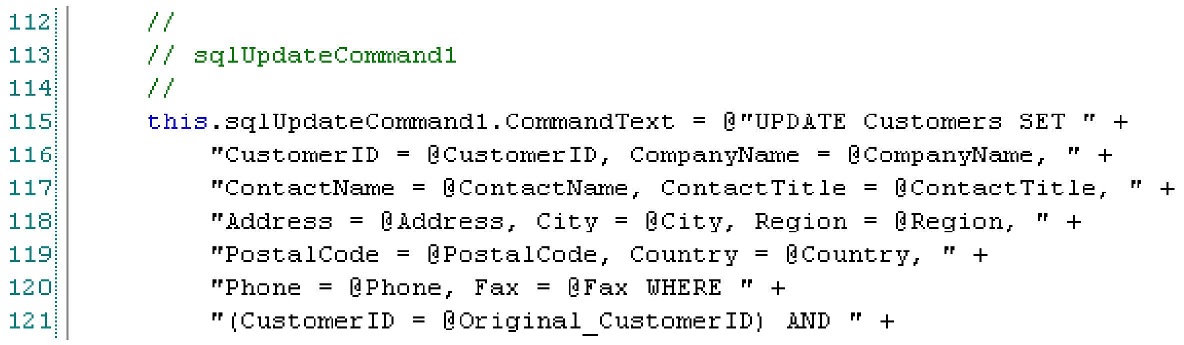
Column	Alias	Table	Output	Sort Type	Sort Order	Criteria
CustomerID		Customers	✓			
CompanyName		Customers	✓			
ContactName		Customers	✓			

```
SELECT     CustomerID, CompanyName, ContactName, ContactTitle, Address, City, Region, PostalCode, Country
FROM       Customers
```

[✓] Regenerate parameters collection for this command.

Picture 2: Visual Studio .NET Query Builder

Query builder’s code format has been designed for machine readability, not for human readability:



```
        //
        // sqlUpdateCommand1
        //
        this.sqlUpdateCommand1.CommandText = @"UPDATE Customers SET " +
            "CustomerID = @CustomerID, CompanyName = @CompanyName, " +
            "ContactName = @ContactName, ContactTitle = @ContactTitle, " +
            "Address = @Address, City = @City, Region = @Region, " +
            "PostalCode = @PostalCode, Country = @Country, " +
            "Phone = @Phone, Fax = @Fax WHERE " +
            "(CustomerID = @Original_CustomerID) AND " +
```

Picture 3: Resulting SQL Code

It turns out that query builder might not be everybody's tool of choice because of other reasons as well. E.g., every additional (or removed) database attribute requires invoking query builder again for code re-generation or even manual code adaptation. There is no syntax highlighting, no testing capability, and query builder tries to optimize logical expressions within SQL, so there is no guarantee hand-coded SQL will stay the way it is.

ADO.NET is also missing support for database independence. Unlike other APIs which offer completely consistent programming interfaces across all database systems and either enforce support for certain SQL standards resp. include semantic extensions in order to address different SQL flavors, ADO.NET providers work in a very vendor-specific manner. While it is true that there are some interface definitions which each provider has to implement (e.g. System.Data.IDbConnection), those interfaces just represent a subset of overall functionality. In .NET 2.0 the ADO.NET provider model has improved, but only at the API level, not at the level of SQL compatibility.

Also, programming against ADO.NET still requires a lot of hand-written code such as creating ParameterCollections, instantiate DataAdapters, Commands and Connections, opening and closing Connections, exception handling, committing or rolling back Transactions, encapsulating database-specifics, and more. While there are third party libraries available covering some of these areas, many of those products either enforce a completely different programming model or involve a steep learning curve. This is where the Vanilla Data Access Layer for .NET steps in.

Introduction to Vanilla DAL

Vanilla DAL is a framework for accessing relational databases with ADO.NET. It avoids the problems that Visual Studio-style rapid application development tends to impose for data access, and at the same time helps improving developer productivity.

Features include:

- SQL statement externalization in XML-files
- Wrapping database-specific code, hence laying the groundwork for database independence
- SQL code generation based on DB or Dataset schema information
- Automatic transaction handling (no need to repeat the same begin transaction-try-commit-catch-rollback sequence over and over again) and transaction propagation on data access
- Optimistic locking without any handwritten code
- SQL statement tracing
- Several convenience functions of common interest

Vanilla is kept simple and allows the programmer to learn and apply the framework within the shortest possible amount of time. Design goals include a lightweight and consistent API, re-use of ADO.NET components (e.g. typed Datasets) and ADO.NET glossary (the main interface IDBAccessor consists of the very method signatures similar to ADO.NET's DataAdapter), close-to-zero performance overhead, and the avoidance of any kind of constraints upon the developer.

Vanilla DAL Tutorial

Vanilla requires a configuration XML-file for each database to be accessed. The configuration schema definition (XSD) makes it easy to let a tool XML-tools auto-generate the basic element structure:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Config" nillable="true" type="Configuration" />
  <xs:complexType name="Configuration">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="1" name="ConnectionString"
type="xs:string" />
      <xs:element minOccurs="1" maxOccurs="1" name="DatabaseType"
type="ConfigDatabaseType" />
      <xs:element minOccurs="0" maxOccurs="1" name="LogSql"
type="xs:boolean" />
      <xs:element minOccurs="0" maxOccurs="1" name="Statements"
type="ArrayOfConfigStatement" />
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="ConfigDatabaseType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Undefined" />
      <xs:enumeration value="SQLServer" />
      <xs:enumeration value="Oracle" />
      <xs:enumeration value="OLEDB" />
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="ArrayOfConfigStatement">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Statement"
nillable="true" type="ConfigStatement" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ConfigStatement">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="1" name="ID" type="xs:string" />
      <xs:element minOccurs="1" maxOccurs="1" name="StatementType"
type="ConfigStatementType" />
      <xs:element minOccurs="0" maxOccurs="1" name="Code" type="xs:string"
/>
      <xs:element minOccurs="0" maxOccurs="1" name="Parameters"
type="ArrayOfConfigParameter" />
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="ConfigStatementType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Undefined" />
      <xs:enumeration value="Text" />
      <xs:enumeration value="StoredProcedure" />
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="ArrayOfConfigParameter">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Parameter"
nillable="true" type="ConfigParameter" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ConfigParameter">
```

```
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="1" name="Name" type="xs:string"
/>
      <xs:element minOccurs="1" maxOccurs="1" name="Type"
type="ConfigParameterType" />
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="ConfigParameterType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Undefined" />
      <xs:enumeration value="Byte" />
      <xs:enumeration value="Int16" />
      <xs:enumeration value="Int32" />
      <xs:enumeration value="Int64" />
      <xs:enumeration value="Double" />
      <xs:enumeration value="Boolean" />
      <xs:enumeration value="DateTime" />
      <xs:enumeration value="String" />
      <xs:enumeration value="Guid" />
      <xs:enumeration value="Decimal" />
      <xs:enumeration value="ByteArray" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

The sample application comes with the following configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- if this is a visual studio embedded resource, recompile project on
each change -->
<Config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <DatabaseType>SQLServer</DatabaseType>
  <LogSql>true</LogSql>
  <!-- Oracle: <DatabaseType>Oracle</DatabaseType> -->
  <!-- OLEDB:  <DatabaseType>OLEDB</DatabaseType> -->
  <Statements>
    <Statement>
      <StatementType>StoredProcedure</StatementType>
      <ID>CustOrderHist</ID>
      <Code>
        <![CDATA[CustOrderHist]]>
      </Code>
      <Parameters>
        <Parameter>
          <Name>customerid</Name>
          <Type>String</Type>
        </Parameter>
      </Parameters>
    </Statement>
    <Statement>
      <StatementType>Text</StatementType>
      <ID>CustomersByCityAndMinOrderCount</ID>
      <Code><![CDATA[
        select  *
        from    customers
        where   city = @city and
                  (select count(*) from orders
                  where orders.customerid = customers.customerid) >=
        @minordercount
      ]]></Code>
      <!-- Oracle:
```

```
      <Code><![CDATA[
        select  *
        from    customers
        where   city = :city and
                  (select count(*) from orders where orders.customerid =
                  customers.customerid) >= :minordercount
      ]]></Code>
      -->
      <!-- OLEDB:
      <Code><![CDATA[
        select  *
        from    customers
        where   city = ? and
                   (select count(*) from orders where orders.customerid =
                   customers.customerid) >= ?
      ]]></Code>
      -->
      <Parameters>
        <Parameter>
          <Name>city</Name>
          <Type>String</Type>
        </Parameter>
        <Parameter>
          <Name>minordercount</Name>
          <Type>Int32</Type>
        </Parameter>
      </Parameters>
    </Statement>
    <Statement>
      <StatementType>Text</StatementType>
      <ID>CustomerCount</ID>
      <Code>
        <![CDATA[
        select  count(*)
        from    customers
      ]]>
      </Code>
      <Parameters>
      </Parameters>
    </Statement>
    <Statement>
      <StatementType>Text</StatementType>
      <ID>DeleteTestCustomers</ID>
      <Code>
        <![CDATA[
        delete
        from    customers
        where   companyname like 'Test%'
      ]]>
      </Code>
      <Parameters>
      </Parameters>
    </Statement>
  </Statements>
</Config>
```

The configuration file basically tells Vanilla which SQL-statements are available for execution. Every SQL-statement can be logged to stdout by setting the `<LogSql>`-element to true.

Additionally Vanilla supports a simple Dataset-to-DB mapping based on ADO.NET Datasets, with no need to hand-code any SQL at all.

Typically the configuration-file will be compiled into the client's assembly. The ADO.NET connection string can either be specified at runtime (see below), or hard-wired inside the configuration file.

The complete Vanilla DAL API is based on the so called IDBAccessor interface. A tangible IDBAccessor implementation is created by the VanillaFactory at startup time:

```
Assembly.GetExecutingAssembly().GetManifestResourceStream(
      "VanillaTest.config.xml"),
      "Data Source=(local);Initial Catalog=Northwind;
      Integrated Security=True");
IDBAccessor accessor = VanillaFactory.CreateDBAccessor(config);
```

Note: SqlServer 2005 Express Edition will install a server instance named (local)\sqlexpress by default.

IDBAccessor is an interface that every database-specific implementation has to support. Working against this interface, the client will never be contaminated with database-specific code. When connecting to another database, all that is required is another configuration file. Multiple configurations can be applied at the same time, simply by instantiating several IDBAccessors.

This is IDBAccessor's current interface (may be subject to change):

```
public interface IDBAccessor {
      IDbCommand CreateCommand(CommandParameter param);
      IDbConnection CreateConnection();
      IDbDataAdapter CreateDataAdapter();

      void Fill(FillParameter param);
      int Update(UpdateParameter param);
      int ExecuteNonQuery(NonQueryParameter param);
      object ExecuteScalar(ScalarParameter param);
      void ExecuteInTransaction(TransactionCallback callback,
      object obj);
}
```

At this point Vanilla is ready to go and may execute its first command (which has been declared in the configuration file - see `<ID>CustomersByCityAndMinOrderCount</ID>`):

```
NorthwindDataset northwindDataset = new NorthwindDataset();
accessor.Fill(new FillParameter(
      northwindDataset.Customers,
      new Statement("CustomersByCityAndMinOrderCount"),
      new ParameterList(
      new Parameter("city", "London"),
      new Parameter("minordercount", 2))
      );
```

The Datatable is now populated with the query's result.

Alternatively, Vanilla DAL can create select-, insert-, update- and delete-statements on-the-fly in case of a 1:1 mapping between Datatable and database. All that needs to be passed in is a Datatable, which will be filled in case of a select, resp. is supposed to hold data for inserts, updates and deletes:

```
northwindDataset.Customers.Clear();
accessor.Fill(new FillParameter(northwindDataset.Customers));
```

The advantage of this approach lies in the fact changes of the underlying database schema do not necessarily require manual SQL code adaptation.

Additionally the `FillParameter.SchemaHandling` property can be applied to define whether the current Datatable schema should be updated by the underlying database schema. By default all columns supported by the Datatable will be fetched from the database (but not more). If there are no Datatable columns, they will be created during runtime. In this case every database column will result in a corresponding column in the Datatable.

Next we will do some in-memory data manipulation, and then update the database accordingly:

```
northwindDataset.Customers.Clear();
accessor.Fill(new FillParameter(northwindDataset.Customers));

foreach (NorthwindDataset.CustomersRow cust1 in
northwindDataset.Customers) {
      cust1.City = "New York";
}
UpdateParameter upd1 =
new UpdateParameter(northwindDataset.Customers);
upd1.RefreshAfterUpdate = true;
upd1.Locking = UpdateParameter.LockingType.Optimistic;
accessor.Update(upd1);
```

`RefreshAfterUpdate = true` tells Vanilla to issue another select command after the update, which is helpful in case database triggers change data during the process of inserting or updateing, or similar. Auto-increment values set by the database are loaded back to the Dataset automatically.

`updateParameter.Locking = UpdateParameter.LockingType.Optimistic` will ensure that only those rows are updated which have not been manipulated by someone else in the meantime. Otherwise a VanillaConcurrencyException will be thrown.

At any time, custom SQL code can be executed as well:

```
// hardcoded custom statement
DataTable table3 = new DataTable();
accessor.Fill(new FillParameter(
            table3,
            new Statement(ConfigStatementType.Text,
            "select * from customers where city = @city"),
            new ParameterList(new Parameter("city", "London"))));
```

Transactional boundaries can be set automatically by invoking ExecuteInTransaction() and providing a TransactionCallback method:

```
// transaction example
northwindDataset.Customers.Clear();
NorthwindDataset.CustomersRow cust2a =
      northwindDataset.Customers.NewCustomersRow();
cust2a.CustomerID = "TEST1";
cust2a.CompanyName = "Tester 1";
northwindDataset.Customers.AddCustomersRow(cust2a);

accessor.ExecuteInTransaction(
new TransactionCallback(DeleteTestCustomers), northwindDataset);

private static void DeleteTestCustomers(IDBAccessor acc,
object param) {
      acc.Update(new UpdateParameter
      (((NorthwindDataset)param).Customers));
      acc.ExecuteNonQuery(new NonQueryParameter(
      new Statement("DeleteTestCustomers")));
}
```

All database commands inside the method run within a single transaction. Any unhandled exception will lead to a rollback.

To sum up, here is complete sample application:

```
try {
    // load db-specific config and instantiate accessor
    VanillaConfig config = VanillaConfig.CreateConfig(

Assembly.GetExecutingAssembly().GetManifestResourceStream("VanillaTest.conf
ig.xml"),
        "Data Source=(local);Initial Catalog=Northwind;Integrated
Security=True");
    // SqlServer: "Data Source=(local);Initial Catalog=Northwind;Integrated
Security=True"
    // SqlServer Express: "Data Source=(local)\\SQLEXPRESS;Initial
Catalog=Northwind;Integrated Security=True"
    // Oracle: "Server=localhost;User ID=scott;Password=tiger"
    // Access: "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\\Program
Files\\Microsoft Office\\OFFICE11\\SAMPLES\\Northwind.mdb"
    IDBAccessor accessor = VanillaFactory.CreateDBAccessor(config);

    // custom statement
    NorthwindDataset northwindDataset = new NorthwindDataset();
    accessor.Fill(new FillParameter(
                      northwindDataset.Customers,
                      new Statement("CustomersByCityAndMinOrderCount"),
                      new ParameterList(
                            new Parameter("city", "London"),
                            new Parameter("minordercount", 2)))
        );

    // insert data
    NorthwindDataset.CustomersRow custIns =
    northwindDataset.Customers.NewCustomersRow();
    custIns.CustomerID = "Foo";
    custIns.CompanyName = "Foo";
    custIns.City = "New York";
    northwindDataset.Customers.AddCustomersRow(custIns);
    try
    {
        accessor.Update(new UpdateParameter(northwindDataset.Customers));
    }
    catch (VanillaException e)
    {
        Console.WriteLine(e.Message);
    }

    // insert and delete data
    NorthwindDataset.CustomersRow custDel =
    northwindDataset.Customers.NewCustomersRow();
    custDel.CustomerID = "Foo2";
    custDel.CompanyName = "Foo2";
    custDel.City = "New York";
    northwindDataset.Customers.AddCustomersRow(custDel);
    try
    {
        accessor.Update(new UpdateParameter(northwindDataset.Customers));
        custDel.Delete();
        accessor.Update(new UpdateParameter(northwindDataset.Customers));
    }
    catch (VanillaException e)
    {
        Console.WriteLine(e.Message);
    }

    // custom statement preprocessed (due to user request)
    northwindDataset.Customers.Clear();
    ConfigStatement stmt =
    config.GetStatement("CustomersByCityAndMinOrderCount");
    ConfigStatement stmt2 =
        new ConfigStatement(stmt.StatementType, stmt.Code + " and 1=1",
        stmt.Parameters);
    accessor.Fill(new FillParameter(
                          northwindDataset.Customers,
                          new Statement(stmt2),
                          new ParameterList(
                                new Parameter("city", "London"),
                                new Parameter("minordercount", 2)))
        );

    // generic statement, this means no sql statement is required (as
dataset maps 1:1 to db)
    // dataset without schema will receive schema information from db (all
columns)
    // FillParameter.SchemaHandling allows to apply different schema
strategies
    DataTable table1 = new DataTable();
    accessor.Fill(new FillParameter(table1, "Customers"));

    // invoke stored procedure
    DataTable table2 = new DataTable();
    accessor.Fill(new FillParameter(
                          table2,
                          new Statement("CustOrderHist"),
                          new ParameterList(new Parameter("customerid",
"Foo"))));

    // hardcoded custom statement
    DataTable table3 = new DataTable();
    accessor.Fill(new FillParameter(
                  table3,
                  new Statement(ConfigStatementType.Text,
                  "select * from customers where city = @city"),
                  new ParameterList(new Parameter("city", "London"))));

    // simulate a concurrency issue
    // fetch the same data twice
    northwindDataset.Customers.Clear();
    // sql code will be generated on-the-fly, based on the columsn defined
in this typed dataset
    accessor.Fill(new FillParameter(northwindDataset.Customers,
    new ParameterList(new Parameter("CustomerID", "Foo"))));

    NorthwindDataset northwindDataset2 = new NorthwindDataset();
    accessor.Fill(new FillParameter(northwindDataset2.Customers,
    new ParameterList(new Parameter("CustomerID", "Foo"))));

    // write some changes back to db
    foreach (NorthwindDataset.CustomersRow cust1 in
    northwindDataset.Customers) {
        cust1.City = "Paris";
    }
    UpdateParameter upd1 = new UpdateParameter(northwindDataset.Customers);
    upd1.RefreshAfterUpdate = true;
    upd1.Locking = UpdateParameter.LockingType.Optimistic;
    accessor.Update(upd1);

    // try to write some other changes back to db which are now based on
wrong original values => concurrency excpetion
    foreach (NorthwindDataset.CustomersRow cust in
    northwindDataset2.Customers) {
        cust.City = "Berlin";
    }
    UpdateParameter upd2 =
    new UpdateParameter(northwindDataset2.Customers);
    upd2.RefreshAfterUpdate = true;
    upd2.Locking = UpdateParameter.LockingType.Optimistic;
    try {
        accessor.Update(upd2);
    }
    catch (VanillaConcurrencyException e) {
        Console.WriteLine(e.Message);
    }

    // transaction example
    northwindDataset.Customers.Clear();
    NorthwindDataset.CustomersRow cust2a =
        northwindDataset.Customers.NewCustomersRow();
    cust2a.CustomerID = "TEST1";
    cust2a.CompanyName = "Tester 1";
    northwindDataset.Customers.AddCustomersRow(cust2a);

    accessor.ExecuteInTransaction(
    new TransactionCallback(DeleteTestCustomers), northwindDataset);

    Console.WriteLine("VanillaTest completed");
}
catch (VanillaException e) {
    Console.WriteLine(e.Message);
}
```

```
private static void DeleteTestCustomers(IDBAccessor acc, object param) {
    acc.Update(new UpdateParameter(((NorthwindDataset)param).Customers));
    acc.ExecuteNonQuery(new NonQueryParameter(new
    Statement("DeleteTestCustomers")));
}
```