ADO to ADO.Net migration strategy

Description

Microsoft incorporated the namespace System.Data.Common in the.Net Framework 2.0, which contains classes intended to be used as the base for all data provider implementation. This architectural decision allows the application designers and programmers to use patterns that are data provider agnostic on their data access layers.

VB6 offers several ways to access a database, all of them using com libraries like ADO and DAO.

The most common structure used to retrieve data from the database in VB6 is the RecordSet. It is a basically a collection of rows retrieved from the database using an specific sql command.

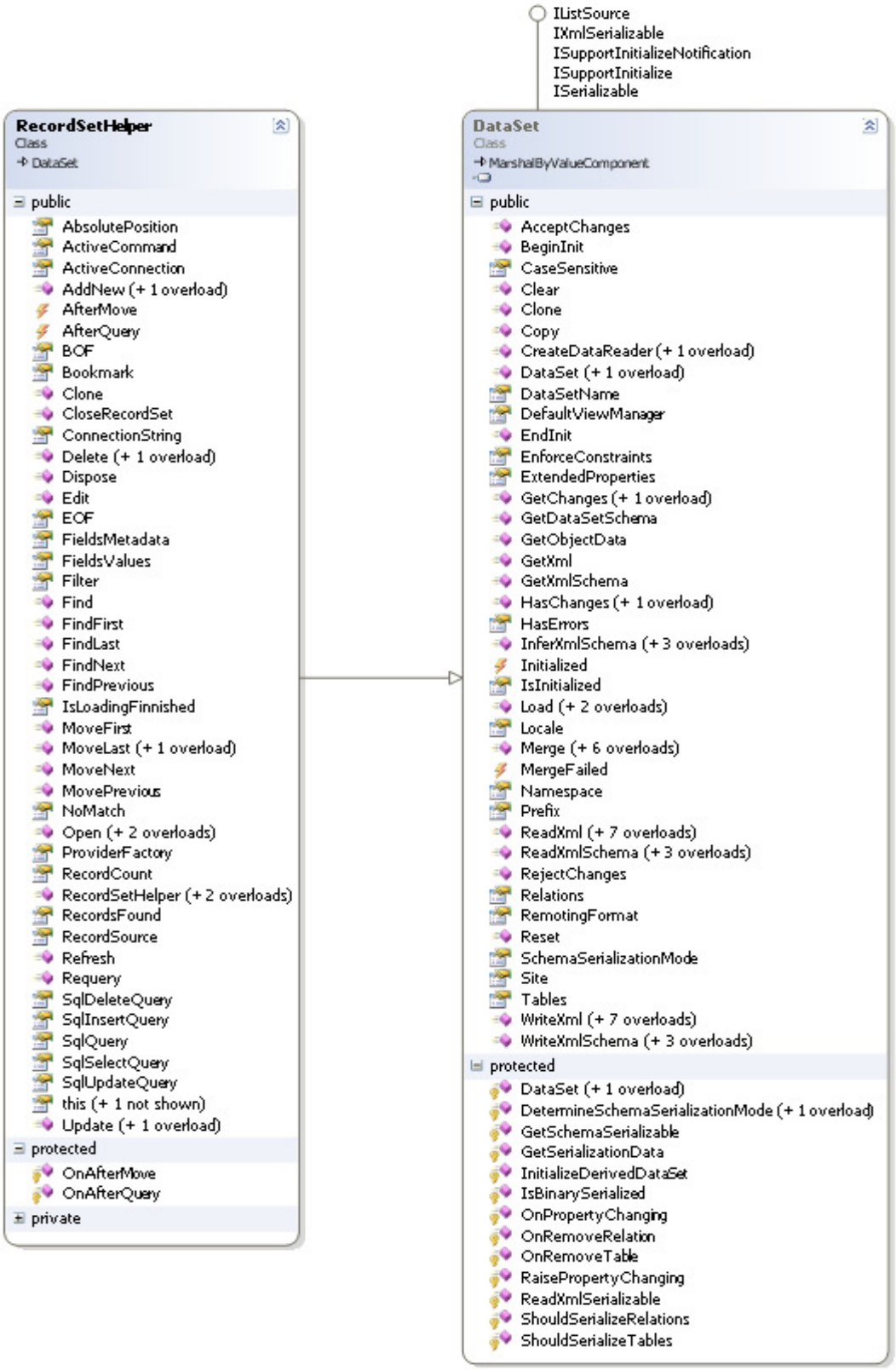
On the .Net side we have the DataSet component, which, like the RecordSet, holds the data retrieved from the database.

There are several differences between those components. The most important difference in terms of functional equivalence is the capability of the recordset to hold the current position and make all the operations on that record.

To accomplish the same functionality in C# .NET, ArtinSoft proposes to develop a Helper class to handle all RecordSet operations. Internally, this class will have all the necessary infrastructure to handle all database requests using DataAdapters, and using only classes from the System.Data.Common namespace.

Helper Class Design

The proposed Helper class will extend the DataSet class, and will give it more functionality without breaking the natural .Net ADO architecture.



The following code is part of the helper code, which is currently under development. This example is for the open methods that encapsulates the dataset population logic.

```
 #region Open Operations
        private void OpenRecordset()
        {
            operationFinished = false;
            DbDataAdapter dbAdapter = CreateAdapter(activeConnection);
            dbAdapter.Fill(this);
            operationFinished = true;
            currentView = Tables[0].DefaultView;
            currentView.AllowDelete = true;
            currentView.AllowEdit = true;
            currentView.AllowNew = true;
            if (Tables[0].Rows.Count == 0)
                index = -1;
            else
                MoveFirst();
            newRow = false;
            foundRecordsCounter = 0;
            OnAfterQuery();
        }

        public void Open()
        {
            if (activeConnection == null && activeCommand != null &&
activeCommand.Connection != null)
               ActiveConnection = activeCommand.Connection;
            else
                throw new InvalidOperationException("The ActiveConnection
property must be set before calling this method");
            OpenRecordset();
        }

        public void Open(DbCommand command, String connectionString)
        {
            this.connectionString = connectionString;
            Open(command, CreateConnection());
        }

        public void Open(DbCommand command, DbConnection connection)
        {
            ActiveConnection = connection;
            activeCommand = command;
            Open();
        }

        private void Open(String SQLstr, String connectionString)
        {
            this.connectionString = connectionString;
            CommandType commandType = getCommandType(SQLstr);
            DbCommand command = providerFactory.CreateCommand();
            command.CommandText = SQLstr;
            command.CommandType = commandType;
            Open(command, connectionString);
        }

        #endregion
```

Advantages

1. Clearer and more readable code in C#. By using this approach there is no need to use generated temporary variables. The generated code will only have a Helper class call.
2. Since the Helper class inherits from the regular ADO.NET DataSet; it would be easy to integrate it with ADO.NET compliant data access frameworks in the future.
3. It minimizes the manual effort required on cases where the RecordSet is handled by more than one function or method with a position dependency.
4. The helper class source code will be included in the deliverables of the project
5. Speeds up the migration project, by minimizing the manual work on the data access code after the automatic migration.`

Examples

Here are some examples of the transformations needed with and without the helper approach

1. Opening a RecordSet

1.1. Source Code

```
Dim mRS As ADODB.Recordset
SQL = ConvertOracleToSqlServer(dbc, SQL)
mRS.Open SQL, dbc, adOpenForwardOnly, adLockReadOnly
```

1.2. Without Helper

```
DataSet mRS = new DataSet();
//We need a provider to create the dataadapter and command
//instances
DbProviderFactory dbProviderFactory =
DbProviderFactories.GetFactory("");
SQL = ConvertOracleToSqlServer(dbc, SQL);
//We need a provider to create a command instance
DbCommand selectcommand = dbProviderFactory.CreateCommand();
selectcommand.CommandType = CommandType.Text
selectcommand.CommandText = SQL
selectcommand.Connection = dbc
//An adapter instance is needed to fill the dataset
DbDataAdapter adapter = dbProviderFactory.CreateDataAdapter();
adapter.SelectCommand = selectcommand;
adapter.Fill(mRS, "table");
```

1.3. With Helper

```
RecordSetHelper mRS = new RecordSetHelper();
SQL = ConvertOracleToSqlServer(dbc, SQL);
//A direct call to the helper class which takes care of the
//adapters and command creations
mRS.Open(SQL, dbc);
```

2. Updating a RecordSet

2.1. Source Code

```
Set rsUpdate.ActiveConnection = dbc
rsUpdate.UpdateBatch
```

2.2. Without Helper

```
//We need a provider to create the dataadapter and command
//instances
DbProviderFactory dbProviderFactory =
DbProviderFactories.GetFactory("");
SQL = ConvertOracleToSqlServer(dbc, SQL);
//We need a provider to create a command instance
//The select command needs to be provided in order to infer the
//proper commands for insertion, deletion and updates
DbCommand selectcommand = dbProviderFactory.CreateCommand();
selectcommand.CommandType = CommandType.Text
selectcommand.CommandText = “Select”
selectcommand.Connection = dbc
//An adapter instance is needed to update the dataset
DbDataAdapter adapter = dbProviderFactory.CreateDataAdapter();
adapter.SelectCommand = selectcommand;
//We need a command builder to infer the proper sql commands
DbCommandBuilder cmdBuilder =
dbProviderFactory.CreateCommandBuilder();
cmdBuilder.DataAdapter = adapter;
adapter.UpdateCommand = cmdBuilder.GetUpdateCommand();
adapter.InsertCommand = cmdBuilder.GetInsertCommand();
adapter.DeleteCommand = cmdBuilder.GetDeleteCommand();
adapter.Update(rsUpdate);
```

2.3. With Helper

```
rsUpdate.ActiveConnection = dbc;
rsUpdate.UpdateBatch();
```

3. RecordSet Iteration

3.1. Source Code

```
With prsChildRecordset
  .Filter = adFilterPendingRecords
  If Not .BOF And Not .EOF Then
    .MoveFirst
    Do Until .EOF
       .Fields(psKey).Value = pvValue
       .Update
       .MoveNext
 Loop
```

3.2. Without Helper

```
//Due to the filter property and value this has to be done
DataTable changedDt = prsChildRecordset.Tables[0].GetChanges();
foreach (DataRow row in changedDt.Rows)
{
   row[psKey] = pvValue;
   row.AcceptChanges();
}
```

3.3. With Helper (not optimized)

```
prsChildRecordset.Filter = adFilterPendingRecords;
if(!adFilterPendingRecords.BOF && !adFilterPendingRecords.EOF)
{
   adFilterPendingRecords.MoveFirst();
   do{
       adFilterPendingRecords[psKey] = pvValue;
       adFilterPendingRecords.Update();
       adFilterPendingRecords.MoveNext();
   }
   while(!adFilterPendingRecords.EOF)
}
```

3.4. With Helper (optimized)

```
//Due to the filter property and value this has to be done
DataTable changedDt = prsChildRecordset.GetChanges();
foreach (DataRow row in changedDt.Rows)
{
   row[psKey] = pvValue;
   row.AcceptChanges();
}
```