```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Linq;
using System.Text;
using System.Xml;

namespace WpfCrudeOperations.ViewModel
{
    public sealed class SqlHelper
    {
        #region private utility methods & constructors
        private SqlHelper() { }
        private static void AttachParameters(SqlCommand command, SqlParameter[]
commandParameters)
        {
            if (command == null) throw new ArgumentNullException("command");
            if (commandParameters != null)
            {
                foreach (SqlParameter p in commandParameters)
                {
                    if (p != null)
                    {
                        // Check for derived output value with no value assigned
                        if ((p.Direction == ParameterDirection.InputOutput ||
                            p.Direction == ParameterDirection.Input) &&
                            (p.Value == null))
                        {
                            p.Value = DBNull.Value;
                        }
                        command.Parameters.Add(p);
                    }
                }
            }
        }
        private static void AssignParameterValues(SqlParameter[] commandParameters,
DataRow dataRow)
        {
            if ((commandParameters == null) || (dataRow == null))
            {
                // Do nothing if we get no data
                return;
            }

            int i = 0;
            // Set the parameters values
            foreach (SqlParameter commandParameter in commandParameters)
            {
                // Check the parameter name
                if (commandParameter.ParameterName == null ||
                    commandParameter.ParameterName.Length <= 1)
                    throw new Exception(
                        string.Format(
                            "Please provide a valid parameter name on the parameter #{0},
the ParameterName property has the following value: '{1}'.",
                            i, commandParameter.ParameterName));
```

```
                if
(dataRow.Table.Columns.IndexOf(commandParameter.ParameterName.Substring(1)) != -1)
                    commandParameter.Value =
dataRow[commandParameter.ParameterName.Substring(1)];
                i++;
            }
        }
        private static void AssignParameterValues(SqlParameter[] commandParameters,
object[] parameterValues)
        {
            if ((commandParameters == null) || (parameterValues == null))
            {
                // Do nothing if we get no data
                return;
            }

            // We must have the same number of values as we pave parameters to put them
in
            if (commandParameters.Length != parameterValues.Length)
            {
                throw new ArgumentException("Parameter count does not match Parameter
Value count.");
            }

            // Iterate through the SqlParameters, assigning the values from the
corresponding position in the
            // value array
            for (int i = 0, j = commandParameters.Length; i < j; i++)
            {
                // If the current array value derives from IDbDataParameter, then assign
its Value property
                if (parameterValues[i] is IDbDataParameter)
                {
                    IDbDataParameter paramInstance =
(IDbDataParameter)parameterValues[i];
                    if (paramInstance.Value == null)
                    {
                        commandParameters[i].Value = DBNull.Value;
                    }
                    else
                    {
                        commandParameters[i].Value = paramInstance.Value;
                    }
                }
                else if (parameterValues[i] == null)
                {
                    commandParameters[i].Value = DBNull.Value;
                }
                else
                {
                    commandParameters[i].Value = parameterValues[i];
                }
            }
        }

        private static void UpdateParameterValues(SqlParameter[] commandParameters,
object[] parameterValues)
        {
```

```
            if ((commandParameters == null) || (parameterValues == null))
            {
                // Do nothing if we get no data
                return;
            }

            // We must have the same number of values as we pave parameters to put them
in
            if (commandParameters.Length != parameterValues.Length)
            {
                throw new ArgumentException("Parameter count does not match Parameter
Value count.");
            }

            // Iterate through the SqlParameters, assigning the values from the
corresponding position in the
            // value array
            for (int i = 0, j = commandParameters.Length; i < j; i++)
            {
                //Update the Return Value
                if (commandParameters[i].Direction == ParameterDirection.ReturnValue)
                {
                    parameterValues[i] = commandParameters[i].Value;
                }

                if (commandParameters[i].Direction == ParameterDirection.InputOutput)
                    parameterValues[i] = commandParameters[i].Value;
            }
        }

        private static void PrepareCommand(SqlCommand command, SqlConnection connection,
SqlTransaction transaction, CommandType commandType, string commandText, SqlParameter[]
commandParameters, out bool mustCloseConnection)
        {
            if (command == null) throw new ArgumentNullException("command");
            if (commandText == null || commandText.Length == 0) throw new
ArgumentNullException("commandText");

            // If the provided connection is not open, we will open it
            if (connection.State != ConnectionState.Open)
            {
                mustCloseConnection = true;
                connection.Open();
            }
            else
            {
                mustCloseConnection = false;
            }

            // Associate the connection with the command
            command.Connection = connection;

            // Set the command text (stored procedure name or SQL statement)
            command.CommandText = commandText;

            // If we were provided a transaction, assign it
            if (transaction != null)
            {
```

```
                if (transaction.Connection == null) throw new ArgumentException("The
transaction was rollbacked or commited, please provide an open transaction.",
"transaction");
                command.Transaction = transaction;
            }

            // Set the command type
            command.CommandType = commandType;

            // Attach the command parameters if they are provided
            if (commandParameters != null)
            {
                AttachParameters(command, commandParameters);
            }
            return;
        }

        #endregion private utility methods & constructors

        #region ExecuteNonQuery
        internal static int ExecuteNonQuery(string connectionString, CommandType
commandType, string commandText)
        {
            try
            {
                // Pass through the call providing null for the set of SqlParameters
                return ExecuteNonQuery(connectionString, commandType, commandText,
(SqlParameter[])null);
            }
            catch (Exception ex)
            {
                throw ex;
            }
        }
        internal static int ExecuteNonQuery(string connectionString, CommandType
commandType, string commandText, params SqlParameter[] commandParameters)
        {
            try
            {
                if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");

                // Create & open a SqlConnection, and dispose of it after we are done
                using (SqlConnection connection = new SqlConnection(connectionString))
                {
                    connection.Open();

                    // Call the overload that takes a connection in place of the
connection string
                    return ExecuteNonQuery(connection, commandType, commandText,
commandParameters);
                }
            }
            catch (Exception ex)
            {
                throw ex;
            }
        }
```

```
        internal static int ExecuteNonQuery(string connectionString, string spName,
params object[] parameterValues)
        {
            try
            {
                int intReturn;
                if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
                if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

                // If we receive parameter values, we need to figure out where they go
                if ((parameterValues != null) && (parameterValues.Length > 0))
                {
                    // Pull the parameters for this stored procedure from the parameter
cache (or discover them & populate the cache)
                    SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connectionString, spName, true);

                    // Assign the provided values to these parameters based on parameter
order
                    AssignParameterValues(commandParameters, parameterValues);

                    // Call the overload that takes an array of SqlParameters
                    //return ExecuteNonQuery(connectionString,
CommandType.StoredProcedure, spName, commandParameters);
                    intReturn = ExecuteNonQuery(connectionString,
CommandType.StoredProcedure, spName, commandParameters);

                    //Update the array -  parameterValues from the new CommandParameters
that should have the ReturnValue
                    UpdateParameterValues(commandParameters, parameterValues);
                    return intReturn;
                }
                else
                {
                    // Otherwise we can just call the SP without params
                    return ExecuteNonQuery(connectionString, CommandType.StoredProcedure,
spName);
                }
            }
            catch (Exception ex)
            {
                throw ex;
            }
        }

        internal static int ExecuteNonQuery(SqlConnection connection, CommandType
commandType, string commandText)
        {
            try
            {
                // Pass through the call providing null for the set of SqlParameters
                return ExecuteNonQuery(connection, commandType, commandText,
(SqlParameter[])null);
            }
            catch (Exception ex)
```

```
            {
                throw ex;
            }
        }

        internal static int ExecuteNonQuery(SqlConnection connection, CommandType
commandType, string commandText, params SqlParameter[] commandParameters)
        {
            try
            {
                if (connection == null) throw new ArgumentNullException("connection");
                // Create a command and prepare it for execution
                SqlCommand cmd = new SqlCommand();
                bool mustCloseConnection = false;
                PrepareCommand(cmd, connection, (SqlTransaction)null, commandType,
commandText, commandParameters, out mustCloseConnection);

                //Added by Deepak Arora on 7-Mar-08
                cmd.CommandTimeout = 0;
                // Finally, execute the command
                int retval = cmd.ExecuteNonQuery();

                // Detach the SqlParameters from the command object, so they can be used
again
                cmd.Parameters.Clear();
                if (mustCloseConnection)
                    connection.Close();
                return retval;
            }
            catch (Exception ex)
            {
                throw ex;
            }
        }

        internal static int ExecuteNonQuery(SqlConnection connection, string spName,
params object[] parameterValues)
        {
            try
            {
                if (connection == null) throw new ArgumentNullException("connection");
                if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

                // If we receive parameter values, we need to figure out where they go
                if ((parameterValues != null) && (parameterValues.Length > 0))
                {
                    // Pull the parameters for this stored procedure from the parameter
cache (or discover them & populate the cache)
                    SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connection, spName);

                    // Assign the provided values to these parameters based on parameter
order
                    AssignParameterValues(commandParameters, parameterValues);

                    // Call the overload that takes an array of SqlParameters
```

```
                    return ExecuteNonQuery(connection, CommandType.StoredProcedure,
spName, commandParameters);
                }
                else
                {
                    // Otherwise we can just call the SP without params
                    return ExecuteNonQuery(connection, CommandType.StoredProcedure,
spName);
                }
            }
            catch (Exception ex)
            {
                throw ex;
            }
        }

        internal static int ExecuteNonQuery(SqlTransaction transaction, CommandType
commandType, string commandText)
        {
            try
            {
                // Pass through the call providing null for the set of SqlParameters
                return ExecuteNonQuery(transaction, commandType, commandText,
(SqlParameter[])null);
            }
            catch (Exception ex)
            {
                throw ex;
            }
        }

        internal static int ExecuteNonQuery(SqlTransaction transaction, CommandType
commandType, string commandText, params SqlParameter[] commandParameters)
        {
            try
            {
                if (transaction == null) throw new ArgumentNullException("transaction");
                if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");

                // Create a command and prepare it for execution
                SqlCommand cmd = new SqlCommand();
                bool mustCloseConnection = false;
                PrepareCommand(cmd, transaction.Connection, transaction, commandType,
commandText, commandParameters, out mustCloseConnection);

                // Finally, execute the command
                int retval = cmd.ExecuteNonQuery();

                // Detach the SqlParameters from the command object, so they can be used
again
                cmd.Parameters.Clear();
                return retval;
            }
            catch (Exception ex)
            {
                throw ex;
```

```
            }
        }

        internal static int ExecuteNonQuery(SqlTransaction transaction, string spName,
params object[] parameterValues)
        {
            try
            {
                if (transaction == null) throw new ArgumentNullException("transaction");
                if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");
                if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

                // If we receive parameter values, we need to figure out where they go
                if ((parameterValues != null) && (parameterValues.Length > 0))
                {
                    // Pull the parameters for this stored procedure from the parameter
cache (or discover them & populate the cache)
                    SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(transaction.Connection, spName);

                    // Assign the provided values to these parameters based on parameter
order
                    AssignParameterValues(commandParameters, parameterValues);

                    // Call the overload that takes an array of SqlParameters
                    return ExecuteNonQuery(transaction, CommandType.StoredProcedure,
spName, commandParameters);
                }
                else
                {
                    // Otherwise we can just call the SP without params
                    return ExecuteNonQuery(transaction, CommandType.StoredProcedure,
spName);
                }
            }
            catch (Exception ex)
            {
                throw ex;
            }
        }

        #endregion ExecuteNonQuery

        #region ExecuteDataset

        public static DataSet ExecuteDataset(string connectionString, CommandType
commandType, string commandText)
        {
            // Pass through the call providing null for the set of SqlParameters
            return ExecuteDataset(connectionString, commandType, commandText,
(SqlParameter[])null);
        }

        internal static DataSet ExecuteDataset(string connectionString, CommandType
commandType, string commandText, params SqlParameter[] commandParameters)
```

```
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");

            // Create & open a SqlConnection, and dispose of it after we are done
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();

                // Call the overload that takes a connection in place of the connection
string
                return ExecuteDataset(connection, commandType, commandText,
commandParameters);
            }
        }

        internal static DataSet ExecuteDataset(string connectionString, string spName,
params object[] parameterValues)
        {
            DataSet dsReturn;
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                //SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connectionString, spName); -- Original code
from sqlHelper
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connectionString, spName, true); // Added
Parameter true to support ReturnValues

                // Assign the provided values to these parameters based on parameter
order
                AssignParameterValues(commandParameters, parameterValues);

                // Call the overload that takes an array of SqlParameters
                //return ExecuteDataset(connectionString, CommandType.StoredProcedure,
spName, commandParameters);
                //Modify code - just store the dataset to dsReturn
                dsReturn = ExecuteDataset(connectionString, CommandType.StoredProcedure,
spName, commandParameters);

                //Update the array -  parameterValues from the new CommandParameters that
should have the ReturnValue
                UpdateParameterValues(commandParameters, parameterValues);
            }
            else
            {
                // Otherwise we can just call the SP without params
                //return ExecuteDataset(connectionString, CommandType.StoredProcedure,
spName);
                //Modify code
```

```
                dsReturn = ExecuteDataset(connectionString, CommandType.StoredProcedure,
spName);
            }
            //Modify code
            return dsReturn;
        }

        internal static DataSet ExecuteDataset(SqlConnection connection, CommandType
commandType, string commandText)
        {
            // Pass through the call providing null for the set of SqlParameters
            return ExecuteDataset(connection, commandType, commandText,
(SqlParameter[])null);
        }

        internal static DataSet ExecuteDataset(SqlConnection connection, CommandType
commandType, string commandText, params SqlParameter[] commandParameters)
        {
            if (connection == null) throw new ArgumentNullException("connection");

            // Create a command and prepare it for execution
            SqlCommand cmd = new SqlCommand();
            bool mustCloseConnection = false;
            PrepareCommand(cmd, connection, (SqlTransaction)null, commandType,
commandText, commandParameters, out mustCloseConnection);

            // Create the DataAdapter & DataSet
            // added by Deepak Arora on 18 Jan 2008
            cmd.CommandTimeout = 200;
            using (SqlDataAdapter da = new SqlDataAdapter(cmd))
            {
                DataSet ds = new DataSet();

                // Fill the DataSet using default values for DataTable names, etc
                da.Fill(ds);

                // Detach the SqlParameters from the command object, so they can be used
again
                cmd.Parameters.Clear();

                if (mustCloseConnection)
                    connection.Close();

                // Return the dataset
                return ds;
            }
        }

        internal static DataSet ExecuteDataset(SqlConnection connection, string spName,
params object[] parameterValues)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
```

```
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connection, spName);

                // Assign the provided values to these parameters based on parameter
order
                AssignParameterValues(commandParameters, parameterValues);

                // Call the overload that takes an array of SqlParameters
                return ExecuteDataset(connection, CommandType.StoredProcedure, spName,
commandParameters);
            }
            else
            {
                // Otherwise we can just call the SP without params
                return ExecuteDataset(connection, CommandType.StoredProcedure, spName);
            }
        }

        internal static DataSet ExecuteDataset(SqlTransaction transaction, CommandType
commandType, string commandText)
        {
            // Pass through the call providing null for the set of SqlParameters
            return ExecuteDataset(transaction, commandType, commandText,
(SqlParameter[])null);
        }

        internal static DataSet ExecuteDataset(SqlTransaction transaction, CommandType
commandType, string commandText, params SqlParameter[] commandParameters)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");

            // Create a command and prepare it for execution
            SqlCommand cmd = new SqlCommand();
            bool mustCloseConnection = false;
            PrepareCommand(cmd, transaction.Connection, transaction, commandType,
commandText, commandParameters, out mustCloseConnection);

            // Create the DataAdapter & DataSet
            using (SqlDataAdapter da = new SqlDataAdapter(cmd))
            {
                DataSet ds = new DataSet();

                // Fill the DataSet using default values for DataTable names, etc
                da.Fill(ds);

                // Detach the SqlParameters from the command object, so they can be used
again
                cmd.Parameters.Clear();

                // Return the dataset
                return ds;
            }
        }
```

```
        internal static DataSet ExecuteDataset(SqlTransaction transaction, string spName,
params object[] parameterValues)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(transaction.Connection, spName);

                // Assign the provided values to these parameters based on parameter
order
                AssignParameterValues(commandParameters, parameterValues);

                // Call the overload that takes an array of SqlParameters
                return ExecuteDataset(transaction, CommandType.StoredProcedure, spName,
commandParameters);
            }
            else
            {
                // Otherwise we can just call the SP without params
                return ExecuteDataset(transaction, CommandType.StoredProcedure, spName);
            }
        }

        #endregion ExecuteDataset

        #region ExecuteReader

        private enum SqlConnectionOwnership
        {
            /// <summary>Connection is owned and managed by SqlHelper</summary>
            Internal,
            /// <summary>Connection is owned and managed by the caller</summary>
            External
        }

        private static SqlDataReader ExecuteReader(SqlConnection connection,
SqlTransaction transaction, CommandType commandType, string commandText, SqlParameter[]
commandParameters, SqlConnectionOwnership connectionOwnership)
        {
            if (connection == null) throw new ArgumentNullException("connection");

            bool mustCloseConnection = false;
            // Create a command and prepare it for execution
            SqlCommand cmd = new SqlCommand();
            try
            {
```

```
                PrepareCommand(cmd, connection, transaction, commandType, commandText,
commandParameters, out mustCloseConnection);

                // Create a reader
                SqlDataReader dataReader;

                // Call ExecuteReader with the appropriate CommandBehavior
                if (connectionOwnership == SqlConnectionOwnership.External)
                {
                    dataReader = cmd.ExecuteReader();
                }
                else
                {
                    dataReader = cmd.ExecuteReader(CommandBehavior.CloseConnection);
                }

                //We need to Close the DataReader in order to get the Return value
                //if (dataReader.IsClosed == false)
                //    dataReader.Close();

                // Detach the SqlParameters from the command object, so they can be used
again.
                // HACK: There is a problem here, the output parameter values are
fletched
                // when the reader is closed, so if the parameters are detached from the
command
                // then the SqlReader can´t set its values.
                // When this happen, the parameters can´t be used again in other command.
                bool canClear = true;
                foreach (SqlParameter commandParameter in cmd.Parameters)
                {
                    if (commandParameter.Direction != ParameterDirection.Input)
                        canClear = false;
                }

                if (canClear)
                {
                    cmd.Parameters.Clear();
                }

                return dataReader;
            }
            catch
            {
                if (mustCloseConnection)
                    connection.Close();
                throw;
            }
        }

        internal static SqlDataReader ExecuteReader(string connectionString, CommandType
commandType, string commandText)
        {
            // Pass through the call providing null for the set of SqlParameters
            return ExecuteReader(connectionString, commandType, commandText,
(SqlParameter[])null);
        }
```

```
        internal static SqlDataReader ExecuteReader(string connectionString, CommandType
commandType, string commandText, params SqlParameter[] commandParameters)
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            SqlConnection connection = null;
            try
            {
                connection = new SqlConnection(connectionString);
                connection.Open();

                // Call the private overload that takes an internally owned connection in
place of the connection string
                return ExecuteReader(connection, null, commandType, commandText,
commandParameters, SqlConnectionOwnership.Internal); //change From Internal to External
            }
            catch
            {
                // If we fail to return the SqlDatReader, we need to close the connection
ourselves
                if (connection != null) connection.Close();
                throw;
            }
        }

        internal static SqlDataReader ExecuteReader(string connectionString, string
spName, params object[] parameterValues)
        {
            SqlDataReader drReturn;
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connectionString, spName, true);

                AssignParameterValues(commandParameters, parameterValues);

                //return ExecuteReader(connectionString, CommandType.StoredProcedure,
spName, commandParameters);
                drReturn = ExecuteReader(connectionString, CommandType.StoredProcedure,
spName, commandParameters);



                // if (drReturn.IsClosed == false)
                //     drReturn.Close();

                UpdateParameterValues(commandParameters, parameterValues);
                return drReturn;
            }
            else
            {
                // Otherwise we can just call the SP without params
```

```
                //return ExecuteReader(connectionString, CommandType.StoredProcedure,
spName);
                drReturn = ExecuteReader(connectionString, CommandType.StoredProcedure,
spName);
                return drReturn;
            }



        }

        internal static SqlDataReader ExecuteReader(SqlConnection connection, CommandType
commandType, string commandText)
        {
            // Pass through the call providing null for the set of SqlParameters
            return ExecuteReader(connection, commandType, commandText,
(SqlParameter[])null);
        }

        internal static SqlDataReader ExecuteReader(SqlConnection connection, CommandType
commandType, string commandText, params SqlParameter[] commandParameters)
        {
            // Pass through the call to the private overload using a null transaction
value and an externally owned connection
            return ExecuteReader(connection, (SqlTransaction)null, commandType,
commandText, commandParameters, SqlConnectionOwnership.External);
        }

        internal static SqlDataReader ExecuteReader(SqlConnection connection, string
spName, params object[] parameterValues)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connection, spName);

                AssignParameterValues(commandParameters, parameterValues);

                return ExecuteReader(connection, CommandType.StoredProcedure, spName,
commandParameters);
            }
            else
            {
                // Otherwise we can just call the SP without params
                return ExecuteReader(connection, CommandType.StoredProcedure, spName);
            }
        }

        internal static SqlDataReader ExecuteReader(SqlTransaction transaction,
CommandType commandType, string commandText)
        {
            // Pass through the call providing null for the set of SqlParameters
```

```
            return ExecuteReader(transaction, commandType, commandText,
(SqlParameter[])null);
        }

        internal static SqlDataReader ExecuteReader(SqlTransaction transaction,
CommandType commandType, string commandText, params SqlParameter[] commandParameters)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");

            // Pass through to private overload, indicating that the connection is owned
by the caller
            return ExecuteReader(transaction.Connection, transaction, commandType,
commandText, commandParameters, SqlConnectionOwnership.External);
        }

        internal static SqlDataReader ExecuteReader(SqlTransaction transaction, string
spName, params object[] parameterValues)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(transaction.Connection, spName);

                AssignParameterValues(commandParameters, parameterValues);

                return ExecuteReader(transaction, CommandType.StoredProcedure, spName,
commandParameters);
            }
            else
            {
                // Otherwise we can just call the SP without params
                return ExecuteReader(transaction, CommandType.StoredProcedure, spName);
            }
        }

        #endregion ExecuteReader

        #region ExecuteScalar

        internal static object ExecuteScalar(string connectionString, CommandType
commandType, string commandText)
        {
            // Pass through the call providing null for the set of SqlParameters
            return ExecuteScalar(connectionString, commandType, commandText,
(SqlParameter[])null);
        }
```

```
        internal static object ExecuteScalar(string connectionString, CommandType
commandType, string commandText, params SqlParameter[] commandParameters)
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            // Create & open a SqlConnection, and dispose of it after we are done
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();

                // Call the overload that takes a connection in place of the connection
string
                return ExecuteScalar(connection, commandType, commandText,
commandParameters);
            }
        }

        internal static object ExecuteScalar(string connectionString, string spName,
params object[] parameterValues)
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connectionString, spName);

                // Assign the provided values to these parameters based on parameter
order
                AssignParameterValues(commandParameters, parameterValues);

                // Call the overload that takes an array of SqlParameters
                return ExecuteScalar(connectionString, CommandType.StoredProcedure,
spName, commandParameters);
            }
            else
            {
                // Otherwise we can just call the SP without params
                return ExecuteScalar(connectionString, CommandType.StoredProcedure,
spName);
            }
        }

        internal static object ExecuteScalar(SqlConnection connection, CommandType
commandType, string commandText)
        {
            // Pass through the call providing null for the set of SqlParameters
            return ExecuteScalar(connection, commandType, commandText,
(SqlParameter[])null);
        }
```

```
        internal static object ExecuteScalar(SqlConnection connection, CommandType
commandType, string commandText, params SqlParameter[] commandParameters)
        {
            if (connection == null) throw new ArgumentNullException("connection");

            // Create a command and prepare it for execution
            SqlCommand cmd = new SqlCommand();

            bool mustCloseConnection = false;
            PrepareCommand(cmd, connection, (SqlTransaction)null, commandType,
commandText, commandParameters, out mustCloseConnection);

            // Execute the command & return the results
            object retval = cmd.ExecuteScalar();

            // Detach the SqlParameters from the command object, so they can be used
again
            cmd.Parameters.Clear();

            if (mustCloseConnection)
                connection.Close();

            return retval;
        }

        internal static object ExecuteScalar(SqlConnection connection, string spName,
params object[] parameterValues)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connection, spName);

                // Assign the provided values to these parameters based on parameter
order
                AssignParameterValues(commandParameters, parameterValues);

                // Call the overload that takes an array of SqlParameters
                return ExecuteScalar(connection, CommandType.StoredProcedure, spName,
commandParameters);
            }
            else
            {
                // Otherwise we can just call the SP without params
                return ExecuteScalar(connection, CommandType.StoredProcedure, spName);
            }
        }

        internal static object ExecuteScalar(SqlTransaction transaction, CommandType
commandType, string commandText)
        {
```

```
            // Pass through the call providing null for the set of SqlParameters
            return ExecuteScalar(transaction, commandType, commandText,
(SqlParameter[])null);
        }

        internal static object ExecuteScalar(SqlTransaction transaction, CommandType
commandType, string commandText, params SqlParameter[] commandParameters)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");

            // Create a command and prepare it for execution
            SqlCommand cmd = new SqlCommand();
            bool mustCloseConnection = false;
            PrepareCommand(cmd, transaction.Connection, transaction, commandType,
commandText, commandParameters, out mustCloseConnection);

            // Execute the command & return the results
            object retval = cmd.ExecuteScalar();

            // Detach the SqlParameters from the command object, so they can be used
again
            cmd.Parameters.Clear();
            return retval;
        }

        internal static object ExecuteScalar(SqlTransaction transaction, string spName,
params object[] parameterValues)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                // PPull the parameters for this stored procedure from the parameter
cache (or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(transaction.Connection, spName);

                // Assign the provided values to these parameters based on parameter
order
                AssignParameterValues(commandParameters, parameterValues);

                // Call the overload that takes an array of SqlParameters
                return ExecuteScalar(transaction, CommandType.StoredProcedure, spName,
commandParameters);
            }
            else
            {
                // Otherwise we can just call the SP without params
                return ExecuteScalar(transaction, CommandType.StoredProcedure, spName);
```

```
            }
        }

        #endregion ExecuteScalar

        #region ExecuteXmlReader
        internal static XmlReader ExecuteXmlReader(SqlConnection connection, CommandType
commandType, string commandText)
        {
            // Pass through the call providing null for the set of SqlParameters
            return ExecuteXmlReader(connection, commandType, commandText,
(SqlParameter[])null);
        }

        internal static XmlReader ExecuteXmlReader(SqlConnection connection, CommandType
commandType, string commandText, params SqlParameter[] commandParameters)
        {
            if (connection == null) throw new ArgumentNullException("connection");

            bool mustCloseConnection = false;
            // Create a command and prepare it for execution
            SqlCommand cmd = new SqlCommand();
            try
            {
                PrepareCommand(cmd, connection, (SqlTransaction)null, commandType,
commandText, commandParameters, out mustCloseConnection);

                // Create the DataAdapter & DataSet
                XmlReader retval = cmd.ExecuteXmlReader();

                // Detach the SqlParameters from the command object, so they can be used
again
                cmd.Parameters.Clear();

                return retval;
            }
            catch
            {
                if (mustCloseConnection)
                    connection.Close();
                throw;
            }
        }

        internal static XmlReader ExecuteXmlReader(SqlConnection connection, string
spName, params object[] parameterValues)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connection, spName);
```

```
                // Assign the provided values to these parameters based on parameter
order
                AssignParameterValues(commandParameters, parameterValues);

                // Call the overload that takes an array of SqlParameters
                return ExecuteXmlReader(connection, CommandType.StoredProcedure, spName,
commandParameters);
            }
            else
            {
                // Otherwise we can just call the SP without params
                return ExecuteXmlReader(connection, CommandType.StoredProcedure, spName);
            }
        }

        internal static XmlReader ExecuteXmlReader(SqlTransaction transaction,
CommandType commandType, string commandText)
        {
            // Pass through the call providing null for the set of SqlParameters
            return ExecuteXmlReader(transaction, commandType, commandText,
(SqlParameter[])null);
        }

        internal static XmlReader ExecuteXmlReader(SqlTransaction transaction,
CommandType commandType, string commandText, params SqlParameter[] commandParameters)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");

            // Create a command and prepare it for execution
            SqlCommand cmd = new SqlCommand();
            bool mustCloseConnection = false;
            PrepareCommand(cmd, transaction.Connection, transaction, commandType,
commandText, commandParameters, out mustCloseConnection);

            // Create the DataAdapter & DataSet
            XmlReader retval = cmd.ExecuteXmlReader();

            // Detach the SqlParameters from the command object, so they can be used
again
            cmd.Parameters.Clear();
            return retval;
        }

        internal static XmlReader ExecuteXmlReader(SqlTransaction transaction, string
spName, params object[] parameterValues)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
```

```
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(transaction.Connection, spName);

                // Assign the provided values to these parameters based on parameter
order
                AssignParameterValues(commandParameters, parameterValues);

                // Call the overload that takes an array of SqlParameters
                return ExecuteXmlReader(transaction, CommandType.StoredProcedure, spName,
commandParameters);
            }
            else
            {
                // Otherwise we can just call the SP without params
                return ExecuteXmlReader(transaction, CommandType.StoredProcedure,
spName);
            }
        }

        #endregion ExecuteXmlReader

        #region FillDataset
        internal static void FillDataset(string connectionString, CommandType
commandType, string commandText, DataSet dataSet, string[] tableNames)
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            if (dataSet == null) throw new ArgumentNullException("dataSet");

            // Create & open a SqlConnection, and dispose of it after we are done
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();

                // Call the overload that takes a connection in place of the connection
string
                FillDataset(connection, commandType, commandText, dataSet, tableNames);
            }
        }

        internal static void FillDataset(string connectionString, CommandType
commandType,
            string commandText, DataSet dataSet, string[] tableNames,
            params SqlParameter[] commandParameters)
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            if (dataSet == null) throw new ArgumentNullException("dataSet");
            // Create & open a SqlConnection, and dispose of it after we are done
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();
```

```
                // Call the overload that takes a connection in place of the connection
string
                FillDataset(connection, commandType, commandText, dataSet, tableNames,
commandParameters);
            }
        }

        internal static void FillDataset(string connectionString, string spName,
            DataSet dataSet, string[] tableNames,
            params object[] parameterValues)
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            if (dataSet == null) throw new ArgumentNullException("dataSet");
            // Create & open a SqlConnection, and dispose of it after we are done
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();

                // Call the overload that takes a connection in place of the connection
string
                FillDataset(connection, spName, dataSet, tableNames, parameterValues);
            }
        }

        internal static void FillDataset(SqlConnection connection, CommandType
commandType,
            string commandText, DataSet dataSet, string[] tableNames)
        {
            FillDataset(connection, commandType, commandText, dataSet, tableNames, null);
        }

        internal static void FillDataset(SqlConnection connection, CommandType
commandType,
            string commandText, DataSet dataSet, string[] tableNames,
            params SqlParameter[] commandParameters)
        {
            FillDataset(connection, null, commandType, commandText, dataSet, tableNames,
commandParameters);
        }

        internal static void FillDataset(SqlConnection connection, string spName,
            DataSet dataSet, string[] tableNames,
            params object[] parameterValues)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (dataSet == null) throw new ArgumentNullException("dataSet");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connection, spName);
```

```
                // Assign the provided values to these parameters based on parameter
order
                AssignParameterValues(commandParameters, parameterValues);

                // Call the overload that takes an array of SqlParameters
                FillDataset(connection, CommandType.StoredProcedure, spName, dataSet,
tableNames, commandParameters);
            }
            else
            {
                // Otherwise we can just call the SP without params
                FillDataset(connection, CommandType.StoredProcedure, spName, dataSet,
tableNames);
            }
        }

        internal static void FillDataset(SqlTransaction transaction, CommandType
commandType,
            string commandText,
            DataSet dataSet, string[] tableNames)
        {
            FillDataset(transaction, commandType, commandText, dataSet, tableNames,
null);
        }

        internal static void FillDataset(SqlTransaction transaction, CommandType
commandType,
            string commandText, DataSet dataSet, string[] tableNames,
            params SqlParameter[] commandParameters)
        {
            FillDataset(transaction.Connection, transaction, commandType, commandText,
dataSet, tableNames, commandParameters);
        }

        internal static void FillDataset(SqlTransaction transaction, string spName,
            DataSet dataSet, string[] tableNames,
            params object[] parameterValues)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");
            if (dataSet == null) throw new ArgumentNullException("dataSet");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(transaction.Connection, spName);

                // Assign the provided values to these parameters based on parameter
order
                AssignParameterValues(commandParameters, parameterValues);
```

```
                // Call the overload that takes an array of SqlParameters
                FillDataset(transaction, CommandType.StoredProcedure, spName, dataSet,
tableNames, commandParameters);
            }
            else
            {
                // Otherwise we can just call the SP without params
                FillDataset(transaction, CommandType.StoredProcedure, spName, dataSet,
tableNames);
            }
        }

        private static void FillDataset(SqlConnection connection, SqlTransaction
transaction, CommandType commandType,
            string commandText, DataSet dataSet, string[] tableNames,
            params SqlParameter[] commandParameters)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (dataSet == null) throw new ArgumentNullException("dataSet");

            // Create a command and prepare it for execution
            SqlCommand command = new SqlCommand();
            command.CommandTimeout = 0;
            bool mustCloseConnection = false;
            PrepareCommand(command, connection, transaction, commandType, commandText,
commandParameters, out mustCloseConnection);

            // Create the DataAdapter & DataSet
            using (SqlDataAdapter dataAdapter = new SqlDataAdapter(command))
            {

                // Add the table mappings specified by the user
                if (tableNames != null && tableNames.Length > 0)
                {
                    string tableName = "Table";
                    for (int index = 0; index < tableNames.Length; index++)
                    {
                        if (tableNames[index] == null || tableNames[index].Length == 0)
throw new ArgumentException("The tableNames parameter must contain a list of tables, a
value was provided as null or empty string.", "tableNames");
                        dataAdapter.TableMappings.Add(tableName, tableNames[index]);
                        tableName += (index + 1).ToString();
                    }
                }

                // Fill the DataSet using default values for DataTable names, etc
                dataAdapter.Fill(dataSet);

                // Detach the SqlParameters from the command object, so they can be used
again
                command.Parameters.Clear();
            }

            if (mustCloseConnection)
                connection.Close();
        }
        #endregion
```

```
        #region UpdateDataset
        internal static void UpdateDataset(SqlCommand insertCommand, SqlCommand
deleteCommand, SqlCommand updateCommand, DataSet dataSet, string tableName)
        {
            if (insertCommand == null) throw new ArgumentNullException("insertCommand");
            if (deleteCommand == null) throw new ArgumentNullException("deleteCommand");
            if (updateCommand == null) throw new ArgumentNullException("updateCommand");
            if (tableName == null || tableName.Length == 0) throw new
ArgumentNullException("tableName");

            // Create a SqlDataAdapter, and dispose of it after we are done
            using (SqlDataAdapter dataAdapter = new SqlDataAdapter())
            {
                // Set the data adapter commands
                dataAdapter.UpdateCommand = updateCommand;
                dataAdapter.InsertCommand = insertCommand;
                dataAdapter.DeleteCommand = deleteCommand;

                // Update the dataset changes in the data source
                dataAdapter.Update(dataSet, tableName);

                // Commit all the changes made to the DataSet
                dataSet.AcceptChanges();
            }
        }
        #endregion

        #region CreateCommand
        internal static SqlCommand CreateCommand(SqlConnection connection, string spName,
params string[] sourceColumns)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // Create a SqlCommand
            SqlCommand cmd = new SqlCommand(spName, connection);
            cmd.CommandType = CommandType.StoredProcedure;

            // If we receive parameter values, we need to figure out where they go
            if ((sourceColumns != null) && (sourceColumns.Length > 0))
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connection, spName);

                // Assign the provided source columns to these parameters based on
parameter order
                for (int index = 0; index < sourceColumns.Length; index++)
                    commandParameters[index].SourceColumn = sourceColumns[index];

                // Attach the discovered parameters to the SqlCommand object
                AttachParameters(cmd, commandParameters);
            }

            return cmd;
        }
```

```
        #endregion

        #region ExecuteNonQueryTypedParams
        internal static int ExecuteNonQueryTypedParams(String connectionString, String
spName, DataRow dataRow)
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If the row has values, the store procedure parameters must be initialized
            if (dataRow != null && dataRow.ItemArray.Length > 0)
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connectionString, spName);

                // Set the parameters values
                AssignParameterValues(commandParameters, dataRow);

                return SqlHelper.ExecuteNonQuery(connectionString,
CommandType.StoredProcedure, spName, commandParameters);
            }
            else
            {
                return SqlHelper.ExecuteNonQuery(connectionString,
CommandType.StoredProcedure, spName);
            }
        }

        internal static int ExecuteNonQueryTypedParams(SqlConnection connection, String
spName, DataRow dataRow)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If the row has values, the store procedure parameters must be initialized
            if (dataRow != null && dataRow.ItemArray.Length > 0)
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connection, spName);

                // Set the parameters values
                AssignParameterValues(commandParameters, dataRow);

                return SqlHelper.ExecuteNonQuery(connection, CommandType.StoredProcedure,
spName, commandParameters);
            }
            else
            {
                return SqlHelper.ExecuteNonQuery(connection, CommandType.StoredProcedure,
spName);
            }
```

```
        }

        internal static int ExecuteNonQueryTypedParams(SqlTransaction transaction, String
spName, DataRow dataRow)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // Sf the row has values, the store procedure parameters must be initialized
            if (dataRow != null && dataRow.ItemArray.Length > 0)
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(transaction.Connection, spName);

                // Set the parameters values
                AssignParameterValues(commandParameters, dataRow);

                return SqlHelper.ExecuteNonQuery(transaction,
CommandType.StoredProcedure, spName, commandParameters);
            }
            else
            {
                return SqlHelper.ExecuteNonQuery(transaction,
CommandType.StoredProcedure, spName);
            }
        }
        #endregion

        #region ExecuteDatasetTypedParams
        internal static DataSet ExecuteDatasetTypedParams(string connectionString, String
spName, DataRow dataRow)
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            //If the row has values, the store procedure parameters must be initialized
            if (dataRow != null && dataRow.ItemArray.Length > 0)
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connectionString, spName);

                // Set the parameters values
                AssignParameterValues(commandParameters, dataRow);

                return SqlHelper.ExecuteDataset(connectionString,
CommandType.StoredProcedure, spName, commandParameters);
            }
            else
```

```
            {
                return SqlHelper.ExecuteDataset(connectionString,
CommandType.StoredProcedure, spName);
            }
        }

        internal static DataSet ExecuteDatasetTypedParams(SqlConnection connection,
String spName, DataRow dataRow)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If the row has values, the store procedure parameters must be initialized
            if (dataRow != null && dataRow.ItemArray.Length > 0)
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connection, spName);

                // Set the parameters values
                AssignParameterValues(commandParameters, dataRow);

                return SqlHelper.ExecuteDataset(connection, CommandType.StoredProcedure,
spName, commandParameters);
            }
            else
            {
                return SqlHelper.ExecuteDataset(connection, CommandType.StoredProcedure,
spName);
            }
        }

        internal static DataSet ExecuteDatasetTypedParams(SqlTransaction transaction,
String spName, DataRow dataRow)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If the row has values, the store procedure parameters must be initialized
            if (dataRow != null && dataRow.ItemArray.Length > 0)
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(transaction.Connection, spName);

                // Set the parameters values
                AssignParameterValues(commandParameters, dataRow);

                return SqlHelper.ExecuteDataset(transaction, CommandType.StoredProcedure,
spName, commandParameters);
            }
```

```
            else
            {
                return SqlHelper.ExecuteDataset(transaction, CommandType.StoredProcedure,
spName);
            }
        }

        #endregion

        #region ExecuteReaderTypedParams

        internal static SqlDataReader ExecuteReaderTypedParams(String connectionString,
String spName, DataRow dataRow)
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If the row has values, the store procedure parameters must be initialized
            if (dataRow != null && dataRow.ItemArray.Length > 0)
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connectionString, spName);

                // Set the parameters values
                AssignParameterValues(commandParameters, dataRow);

                return SqlHelper.ExecuteReader(connectionString,
CommandType.StoredProcedure, spName, commandParameters);
            }
            else
            {
                return SqlHelper.ExecuteReader(connectionString,
CommandType.StoredProcedure, spName);
            }
        }



        internal static SqlDataReader ExecuteReaderTypedParams(SqlConnection connection,
String spName, DataRow dataRow)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If the row has values, the store procedure parameters must be initialized
            if (dataRow != null && dataRow.ItemArray.Length > 0)
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connection, spName);

                // Set the parameters values
```

```
                AssignParameterValues(commandParameters, dataRow);

                return SqlHelper.ExecuteReader(connection, CommandType.StoredProcedure,
spName, commandParameters);
            }
            else
            {
                return SqlHelper.ExecuteReader(connection, CommandType.StoredProcedure,
spName);
            }
        }


        internal static SqlDataReader ExecuteReaderTypedParams(SqlTransaction
transaction, String spName, DataRow dataRow)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If the row has values, the store procedure parameters must be initialized
            if (dataRow != null && dataRow.ItemArray.Length > 0)
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(transaction.Connection, spName);

                // Set the parameters values
                AssignParameterValues(commandParameters, dataRow);

                return SqlHelper.ExecuteReader(transaction, CommandType.StoredProcedure,
spName, commandParameters);
            }
            else
            {
                return SqlHelper.ExecuteReader(transaction, CommandType.StoredProcedure,
spName);
            }
        }
        #endregion

        #region ExecuteScalarTypedParams

        internal static object ExecuteScalarTypedParams(String connectionString, String
spName, DataRow dataRow)
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If the row has values, the store procedure parameters must be initialized
            if (dataRow != null && dataRow.ItemArray.Length > 0)
            {
```

```
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connectionString, spName);

                // Set the parameters values
                AssignParameterValues(commandParameters, dataRow);

                return SqlHelper.ExecuteScalar(connectionString,
CommandType.StoredProcedure, spName, commandParameters);
            }
            else
            {
                return SqlHelper.ExecuteScalar(connectionString,
CommandType.StoredProcedure, spName);
            }
        }


        internal static object ExecuteScalarTypedParams(SqlConnection connection, String
spName, DataRow dataRow)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If the row has values, the store procedure parameters must be initialized
            if (dataRow != null && dataRow.ItemArray.Length > 0)
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connection, spName);

                // Set the parameters values
                AssignParameterValues(commandParameters, dataRow);

                return SqlHelper.ExecuteScalar(connection, CommandType.StoredProcedure,
spName, commandParameters);
            }
            else
            {
                return SqlHelper.ExecuteScalar(connection, CommandType.StoredProcedure,
spName);
            }
        }


        internal static object ExecuteScalarTypedParams(SqlTransaction transaction,
String spName, DataRow dataRow)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");
```

```
            // If the row has values, the store procedure parameters must be initialized
            if (dataRow != null && dataRow.ItemArray.Length > 0)
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(transaction.Connection, spName);

                // Set the parameters values
                AssignParameterValues(commandParameters, dataRow);

                return SqlHelper.ExecuteScalar(transaction, CommandType.StoredProcedure,
spName, commandParameters);
            }
            else
            {
                return SqlHelper.ExecuteScalar(transaction, CommandType.StoredProcedure,
spName);
            }
        }
        #endregion

        #region ExecuteXmlReaderTypedParams

        internal static XmlReader ExecuteXmlReaderTypedParams(SqlConnection connection,
String spName, DataRow dataRow)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If the row has values, the store procedure parameters must be initialized
            if (dataRow != null && dataRow.ItemArray.Length > 0)
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connection, spName);

                // Set the parameters values
                AssignParameterValues(commandParameters, dataRow);

                return SqlHelper.ExecuteXmlReader(connection,
CommandType.StoredProcedure, spName, commandParameters);
            }
            else
            {
                return SqlHelper.ExecuteXmlReader(connection,
CommandType.StoredProcedure, spName);
            }
        }


        internal static XmlReader ExecuteXmlReaderTypedParams(SqlTransaction transaction,
String spName, DataRow dataRow)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
```

```
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If the row has values, the store procedure parameters must be initialized
            if (dataRow != null && dataRow.ItemArray.Length > 0)
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(transaction.Connection, spName);

                // Set the parameters values
                AssignParameterValues(commandParameters, dataRow);

                return SqlHelper.ExecuteXmlReader(transaction,
CommandType.StoredProcedure, spName, commandParameters);
            }
            else
            {
                return SqlHelper.ExecuteXmlReader(transaction,
CommandType.StoredProcedure, spName);
            }
        }
        #endregion

        #region ExecuteDataTable


        public static DataTable ExecuteDataTable(string connectionString, CommandType
commandType, string commandText)
        {
            // Pass through the call providing null for the set of SqlParameters
            return ExecuteDataTable(connectionString, commandType, commandText,
(SqlParameter[])null);
        }


        internal static DataTable ExecuteDataTable(string connectionString, CommandType
commandType, string commandText, params SqlParameter[] commandParameters)
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");

            // Create & open a SqlConnection, and dispose of it after we are done
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();

                // Call the overload that takes a connection in place of the connection
string
                return ExecuteDataTable(connection, commandType, commandText,
commandParameters);
            }
        }
```

```
        internal static DataTable ExecuteDataTable(string connectionString, string
spName, params object[] parameterValues)
        {
            DataTable dsReturn;
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                //SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connectionString, spName); -- Original code
from sqlHelper
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connectionString, spName, true); // Added
Parameter true to support ReturnValues

                // Assign the provided values to these parameters based on parameter
order
                AssignParameterValues(commandParameters, parameterValues);

                // Call the overload that takes an array of SqlParameters
                //return ExecuteDataTable(connectionString, CommandType.StoredProcedure,
spName, commandParameters);
                //Modify code - just store the DataTable to dsReturn
                dsReturn = ExecuteDataTable(connectionString,
CommandType.StoredProcedure, spName, commandParameters);

                //Update the array -  parameterValues from the new CommandParameters that
should have the ReturnValue
                UpdateParameterValues(commandParameters, parameterValues);
            }
            else
            {
                // Otherwise we can just call the SP without params
                //return ExecuteDataTable(connectionString, CommandType.StoredProcedure,
spName);
                //Modify code
                dsReturn = ExecuteDataTable(connectionString,
CommandType.StoredProcedure, spName);
            }
            //Modify code
            return dsReturn;
        }


        internal static DataTable ExecuteDataTable(SqlConnection connection, CommandType
commandType, string commandText)
        {
            // Pass through the call providing null for the set of SqlParameters
            return ExecuteDataTable(connection, commandType, commandText,
(SqlParameter[])null);
        }
```

```
        internal static DataTable ExecuteDataTable(SqlConnection connection, CommandType
commandType, string commandText, params SqlParameter[] commandParameters)
        {
            if (connection == null) throw new ArgumentNullException("connection");

            // Create a command and prepare it for execution
            SqlCommand cmd = new SqlCommand();
            bool mustCloseConnection = false;
            PrepareCommand(cmd, connection, (SqlTransaction)null, commandType,
commandText, commandParameters, out mustCloseConnection);

            // Create the DataAdapter & DataTable
            // added by Deepak Arora on 18 Jan 2008
            cmd.CommandTimeout = 200;
            using (SqlDataAdapter da = new SqlDataAdapter(cmd))
            {
                DataTable ds = new DataTable();

                // Fill the DataTable using default values for DataTable names, etc
                da.Fill(ds);

                // Detach the SqlParameters from the command object, so they can be used
again
                cmd.Parameters.Clear();

                if (mustCloseConnection)
                    connection.Close();

                // Return the DataTable
                return ds;
            }
        }


        internal static DataTable ExecuteDataTable(SqlConnection connection, string
spName, params object[] parameterValues)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(connection, spName);

                // Assign the provided values to these parameters based on parameter
order
                AssignParameterValues(commandParameters, parameterValues);

                // Call the overload that takes an array of SqlParameters
                return ExecuteDataTable(connection, CommandType.StoredProcedure, spName,
commandParameters);
```

```
            }
            else
            {
                // Otherwise we can just call the SP without params
                return ExecuteDataTable(connection, CommandType.StoredProcedure, spName);
            }
        }


        internal static DataTable ExecuteDataTable(SqlTransaction transaction,
CommandType commandType, string commandText)
        {
            // Pass through the call providing null for the set of SqlParameters
            return ExecuteDataTable(transaction, commandType, commandText,
(SqlParameter[])null);
        }


        internal static DataTable ExecuteDataTable(SqlTransaction transaction,
CommandType commandType, string commandText, params SqlParameter[] commandParameters)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");

            // Create a command and prepare it for execution
            SqlCommand cmd = new SqlCommand();
            bool mustCloseConnection = false;
            PrepareCommand(cmd, transaction.Connection, transaction, commandType,
commandText, commandParameters, out mustCloseConnection);

            // Create the DataAdapter & DataTable
            using (SqlDataAdapter da = new SqlDataAdapter(cmd))
            {
                DataTable ds = new DataTable();

                // Fill the DataTable using default values for DataTable names, etc
                da.Fill(ds);

                // Detach the SqlParameters from the command object, so they can be used
again
                cmd.Parameters.Clear();

                // Return the DataTable
                return ds;
            }
        }


        internal static DataTable ExecuteDataTable(SqlTransaction transaction, string
spName, params object[] parameterValues)
        {
            if (transaction == null) throw new ArgumentNullException("transaction");
            if (transaction != null && transaction.Connection == null) throw new
ArgumentException("The transaction was rollbacked or commited, please provide an open
transaction.", "transaction");
```

```
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            // If we receive parameter values, we need to figure out where they go
            if ((parameterValues != null) && (parameterValues.Length > 0))
            {
                // Pull the parameters for this stored procedure from the parameter cache
(or discover them & populate the cache)
                SqlParameter[] commandParameters =
SqlHelperParameterCache.GetSpParameterSet(transaction.Connection, spName);

                // Assign the provided values to these parameters based on parameter
order
                AssignParameterValues(commandParameters, parameterValues);

                // Call the overload that takes an array of SqlParameters
                return ExecuteDataTable(transaction, CommandType.StoredProcedure, spName,
commandParameters);
            }
            else
            {
                // Otherwise we can just call the SP without params
                return ExecuteDataTable(transaction, CommandType.StoredProcedure,
spName);
            }
        }
        #endregion ExecuteDataTable
    }
    internal sealed class SqlHelperParameterCache
    {
        #region private methods, variables, and constructors

        //Since this class provides only static methods, make the default constructor
private to prevent
        //instances from being created with "new SqlHelperParameterCache()"
        private SqlHelperParameterCache() { }

        private static Hashtable paramCache = Hashtable.Synchronized(new Hashtable());

        /// <summary>
        /// Resolve at run time the appropriate set of SqlParameters for a stored
procedure
        /// </summary>
        /// <param name="connection">A valid SqlConnection object</param>
        /// <param name="spName">The name of the stored procedure</param>
        /// <param name="includeReturnValueParameter">Whether or not to include their
return value parameter</param>
        /// <returns>The parameter array discovered.</returns>
        private static SqlParameter[] DiscoverSpParameterSet(SqlConnection connection,
string spName, bool includeReturnValueParameter)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            SqlCommand cmd = new SqlCommand(spName, connection);
            cmd.CommandType = CommandType.StoredProcedure;
```

```
            connection.Open();
            SqlCommandBuilder.DeriveParameters(cmd);
            connection.Close();

            if (!includeReturnValueParameter)
            {
                cmd.Parameters.RemoveAt(0);
            }

            SqlParameter[] discoveredParameters = new SqlParameter[cmd.Parameters.Count];

            cmd.Parameters.CopyTo(discoveredParameters, 0);

            // Init the parameters with a DBNull value
            foreach (SqlParameter discoveredParameter in discoveredParameters)
            {
                discoveredParameter.Value = DBNull.Value;
            }
            return discoveredParameters;
        }

        /// <summary>
        /// Deep copy of cached SqlParameter array
        /// </summary>
        /// <param name="originalParameters"></param>
        /// <returns></returns>
        private static SqlParameter[] CloneParameters(SqlParameter[] originalParameters)
        {
            SqlParameter[] clonedParameters = new
SqlParameter[originalParameters.Length];

            for (int i = 0, j = originalParameters.Length; i < j; i++)
            {
                clonedParameters[i] =
(SqlParameter)((ICloneable)originalParameters[i]).Clone();
            }

            return clonedParameters;
        }

        #endregion private methods, variables, and constructors

        #region caching functions

        /// <summary>
        /// Add parameter array to the cache
        /// </summary>
        /// <param name="connectionString">A valid connection string for a
SqlConnection</param>
        /// <param name="commandText">The stored procedure name or T-SQL command</param>
        /// <param name="commandParameters">An array of SqlParamters to be cached</param>
        internal static void CacheParameterSet(string connectionString, string
commandText, params SqlParameter[] commandParameters)
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            if (commandText == null || commandText.Length == 0) throw new
ArgumentNullException("commandText");
```

```
            string hashKey = connectionString + ":" + commandText;

            paramCache[hashKey] = commandParameters;
        }

        /// <summary>
        /// Retrieve a parameter array from the cache
        /// </summary>
        /// <param name="connectionString">A valid connection string for a
SqlConnection</param>
        /// <param name="commandText">The stored procedure name or T-SQL command</param>
        /// <returns>An array of SqlParamters</returns>
        internal static SqlParameter[] GetCachedParameterSet(string connectionString,
string commandText)
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            if (commandText == null || commandText.Length == 0) throw new
ArgumentNullException("commandText");

            string hashKey = connectionString + ":" + commandText;

            SqlParameter[] cachedParameters = paramCache[hashKey] as SqlParameter[];
            if (cachedParameters == null)
            {
                return null;
            }
            else
            {
                return CloneParameters(cachedParameters);
            }
        }

        #endregion caching functions

        #region Parameter Discovery Functions

        /// <summary>
        /// Retrieves the set of SqlParameters appropriate for the stored procedure
        /// </summary>
        /// <remarks>
        /// This method will query the database for this information, and then store it
in a cache for future requests.
        /// </remarks>
        /// <param name="connectionString">A valid connection string for a
SqlConnection</param>
        /// <param name="spName">The name of the stored procedure</param>
        /// <returns>An array of SqlParameters</returns>
        internal static SqlParameter[] GetSpParameterSet(string connectionString, string
spName)
        {
            return GetSpParameterSet(connectionString, spName, false);
        }

        /// <summary>
        /// Retrieves the set of SqlParameters appropriate for the stored procedure
        /// </summary>
```

```
        /// <remarks>
        /// This method will query the database for this information, and then store it
in a cache for future requests.
        /// </remarks>
        /// <param name="connectionString">A valid connection string for a
SqlConnection</param>
        /// <param name="spName">The name of the stored procedure</param>
        /// <param name="includeReturnValueParameter">A bool value indicating whether the
return value parameter should be included in the results</param>
        /// <returns>An array of SqlParameters</returns>
        internal static SqlParameter[] GetSpParameterSet(string connectionString, string
spName, bool includeReturnValueParameter)
        {
            if (connectionString == null || connectionString.Length == 0) throw new
ArgumentNullException("connectionString");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                return GetSpParameterSetInternal(connection, spName,
includeReturnValueParameter);
            }
        }

        /// <summary>
        /// Retrieves the set of SqlParameters appropriate for the stored procedure
        /// </summary>
        /// <remarks>
        /// This method will query the database for this information, and then store it
in a cache for future requests.
        /// </remarks>
        /// <param name="connection">A valid SqlConnection object</param>
        /// <param name="spName">The name of the stored procedure</param>
        /// <returns>An array of SqlParameters</returns>
        internal static SqlParameter[] GetSpParameterSet(SqlConnection connection, string
spName)
        {
            return GetSpParameterSet(connection, spName, false);
        }

        /// <summary>
        /// Retrieves the set of SqlParameters appropriate for the stored procedure
        /// </summary>
        /// <remarks>
        /// This method will query the database for this information, and then store it
in a cache for future requests.
        /// </remarks>
        /// <param name="connection">A valid SqlConnection object</param>
        /// <param name="spName">The name of the stored procedure</param>
        /// <param name="includeReturnValueParameter">A bool value indicating whether the
return value parameter should be included in the results</param>
        /// <returns>An array of SqlParameters</returns>
        internal static SqlParameter[] GetSpParameterSet(SqlConnection connection, string
spName, bool includeReturnValueParameter)
        {
            if (connection == null) throw new ArgumentNullException("connection");
```

```
            using (SqlConnection clonedConnection =
(SqlConnection)((ICloneable)connection).Clone())
            {
                return GetSpParameterSetInternal(clonedConnection, spName,
includeReturnValueParameter);
            }
        }

        /// <summary>
        /// Retrieves the set of SqlParameters appropriate for the stored procedure
        /// </summary>
        /// <param name="connection">A valid SqlConnection object</param>
        /// <param name="spName">The name of the stored procedure</param>
        /// <param name="includeReturnValueParameter">A bool value indicating whether the
return value parameter should be included in the results</param>
        /// <returns>An array of SqlParameters</returns>
        private static SqlParameter[] GetSpParameterSetInternal(SqlConnection connection,
string spName, bool includeReturnValueParameter)
        {
            if (connection == null) throw new ArgumentNullException("connection");
            if (spName == null || spName.Length == 0) throw new
ArgumentNullException("spName");

            string hashKey = connection.ConnectionString + ":" + spName +
(includeReturnValueParameter ? ":include ReturnValue Parameter" : "");

            SqlParameter[] cachedParameters;

            cachedParameters = paramCache[hashKey] as SqlParameter[];
            if (cachedParameters == null)
            {
                SqlParameter[] spParameters = DiscoverSpParameterSet(connection, spName,
includeReturnValueParameter);
                paramCache[hashKey] = spParameters;
                cachedParameters = spParameters;
            }

            return CloneParameters(cachedParameters);
        }




        #endregion Parameter Discovery Functions

    }
}
```