Multi-Level Grid Navigator for Visual Studio .NET

Robert Dollinger
Computing and Information Systems
University of Wisconsin Stevens Point
Stevens Point, WI 54481
rdolling@uwsp.edu

Abstract

Visual Studio .NET came with a versatile DataGrid class that allows navigation across several DataTables of a DataSet connected as a DataSource to the grid. When navigation follows a foreign key connection, only the child rows of the row selected in the parent table are visualized in the grid. In this case the row number of the current row in the grid does not match the current row in the corresponding DataTable and we end up processing the wrong data unless specific code is provided to deal with the situation. We present the concept and implementation of a general purpose GridNavigator that solves the problem of correlating the current row in the grid with the corresponding row in the data table. The GridNavigator uses a stack in order to keep track of the navigation paths across the tables and provides a generalized and systematic approach to the above described problem.

1 Introduction

Visual Studio .NET provides GUI developers with a versatile DataGrid class that allows navigation across several DataTables of a DataSet that is connected as a DataSource to the grid. This is a powerful user interface feature since it allows visualizing the content of several tables while in the same grid by simply navigating from one table to the other. Navigation can be as simple as clicking on the table's name at the root level, in which case the entire content of the selected table is visualized, or can be done following foreign key connections between the tables of the DataSet. In this later case, only some of the rows of the current table are visualized in the grid, namely the child rows of the row selected in the parent table.

In a typical application the user would select one or more of the rows in the grid and interactively perform various operations like changing the values of fields, adding new rows or deleting existing rows. Each and every such operation should accurately translate into a corresponding operation at the level of the underlying DataTable. Unfortunately, this is not happening when the rows visualized in the grid are the subset of child rows of a row previously selected in a parent table. In this case the row the user works on in the grid may not be the same with the one that gets affected in the underlying DataTable, which simply means the user unintentionally ends up working on the wrong data.

Although there is no built in functionality to correlate between what we see and what we process, users can provide their own code to solve this by using existing support like the GetChildRows() method. This assumes keeping track of each navigation step and the corresponding selected parent row. Coding this on an ad-hoc basis can become messy in cases when parent-child navigation goes back and forth across several levels and when the DataSet consists of a larger number of tables with several connections. In this paper we present the concept and implementation of a GridNavigator that solves the problem of correlating the current row in the grid with the corresponding row in the data table. The GridNavigator uses a stack in order to keep track of the navigation paths across the tables and provides a generalized and systematic approach to the above described problem. It works for arbitrary DataSets and arbitrary configurations of foreign key connections between the tables. The GridNavigator has been developed as a .NET class with a very simple interface and compiled as a DLL ready to be included in applications.

2 A Working Example

We use a simple working example in order to illustrate the problems that may emerge when connecting a DataGrid that is visualizing multiple tables of a DataSet connected to it, as well as the solution proposed in this paper. The DataSet we use contains 3 tables: Tb_Supplier, Tb_Offers and Tb_Transactions. The Tb_Offers table contains a Supp_ID column which is declared as a foreign key with name Supplier_Offers, such that values of this column identify the supplier making the current offer. Similarly, the Tb_Transactions table contains a Supp_ID column which is declared as a foreign key with name Supplier_Transactions, such that values of this column identify the supplier for each

transaction. In addition, as shown in figure 1, there is a third foreign key in Tb_Transactions, called Offers_Transactions, composed of attributes Supp_ID and Prod_ID, pointing to the attributes with the same names in Tb_Offers. This connection makes sure that a supplier cannot have a transaction on a product unless he/she is first offering that product.

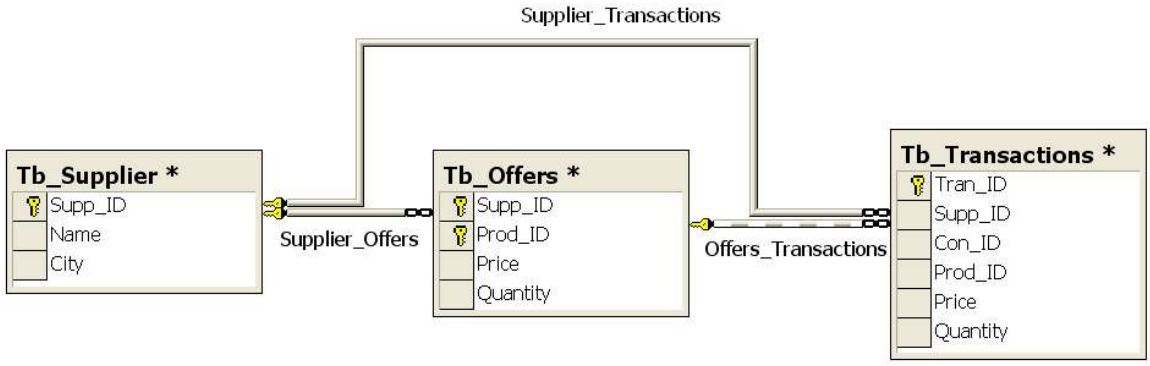

Figure 1: Structure of the sample dataset.

The DataSet is connected as a DataSource to a DataGrid control on a simple Windows form. The form also contains a set of textboxes that are intended to visualize the content of the row currently processed. A couple of buttons for basic operations like: add and delete row complete our testing Windows form.

The grid can visualize the content of any of the 3 tables of the DataSet independently by clicking on the name of the table we choose when at the root level. The text boxes above the grid are supposed to visualize the content of the current row in the grid while moving up and down across the rows of the selected table. In order to achieve this we need to use a property of the DataGrid control called CurrentCell which identifies the active cell in the grid. CurrentCell has at its turn a property called RowNumber giving the number of the current row in the grid. One would use the value of this property in order to programmatically process the corresponding data in the DataTable. Simply put, if we want to visualize in our textboxes the values of the current row in the grid we would assign to the textboxes the values of the row in the DataTable located at the position given by the CurrentCell.RowNumber property. We do this whenever the current cell/row in the grid changes and we will have the textboxes accurately reflecting the values of the row at the current position in the grid as illustrated in figure 2.

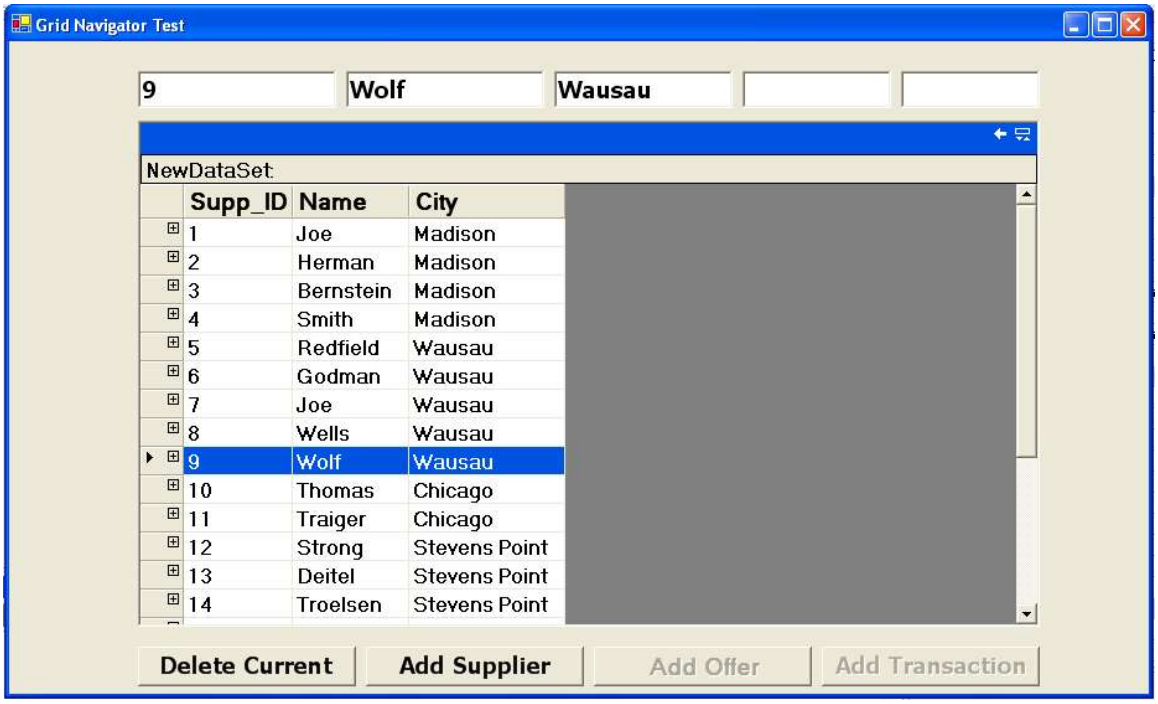

Figure 2: Values in the currently selected row are visualized in the upper textboxes

A nice feature of the DataGrid when connected to a DataSet is that it allows cross table navigation. One can follow any of the table connections provided by the foreign keys in figure 1. From the Tb_Supplier table one can navigate to Tb_Offers to visualize the offers of a selected supplier by following the Supplier_Offers link. If we follow the Supplier_Transactions link we get the transactions of the supplier. In our example one can just expand supplier number 9 and move to the Tb_Offers table by following the Supplier_Offers link. The effect is that the grid will visualize only the subset of offers corresponding to supplier number 9, instead of the entire content of the Tb_Offers table as one can observe in figure 3.

3 The Synchronization Problem

The normal expectation is that at any time one can move from one row to another of those currently visualized in the grid and programmatically perform various processing tasks on the current row (like: delete, update or whatever else). This works fine and in accordance with everyone's expectations as long as the grid is visualizing the rows of a table independently. However, it is not true anymore when the grid visualizes the child rows of a row from a parent table (e.g. the subset of child rows representing the offers of a selected supplier parent row) as in figure 3. In this case the value of the RowNumber property corresponding to the current row in the grid does not match with the right row in the DataTable and we end up processing the wrong data.

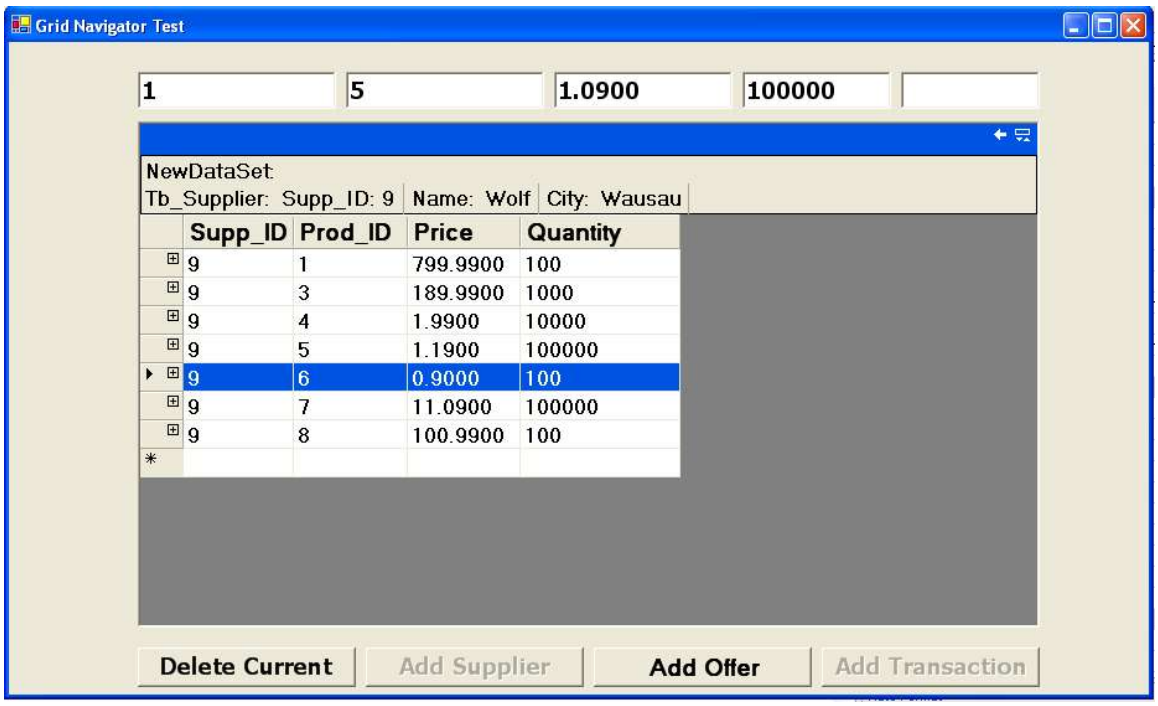

Figure 3: Wrong values are visualized in the upper textboxes when the grid contains the subset of offers corresponding to a previously selected supplier.

In our example from figure 3 the grid contains only a subset of the rows in the Tb_Offers table, namely the offers corresponding to supplier number 9. If we click on the row with Prod_ID=6, the value of the RowNumber property is 4 (fifth row in the grid) and using this value as an index on the DataTable will return the fifth row of the full offers table which happens to be the fifth offer of the first supplier and not one of the offers that are shown in the grid. This is why the values shown in the text boxes end up to be the wrong ones.

3.1 Fixing the Problem

The way the DataGrid control should behave in the case of the previous example is illustrated in figure 4.
Given the inconsistent behavior of the DataGrid control when connected to DataSet, it may be a matter of debate why this problem is not solved in the first place by providing appropriate behavior for the controls involved. It is not realistic to expect this problem to be solved at the level of the DataSet because it's not a data specific problem. One may expect that the above illustrated problem should have been taken care of on the side of the DataGrid control. There could be several reasons why this is not the case: (a) a grid control accepts various types of data sources like: arrays, arraylists, tables etc. The problem appears only in the case of datasets; (b) there are several types of grids, e.g. for Windows Forms and Web Forms, with different implementations, but with a common problem, and this problem may be expected to find its solution outside the

implementation of any grid control. Another, unlikely, explanation could be that Microsoft simply omitted to appropriately address the issue.

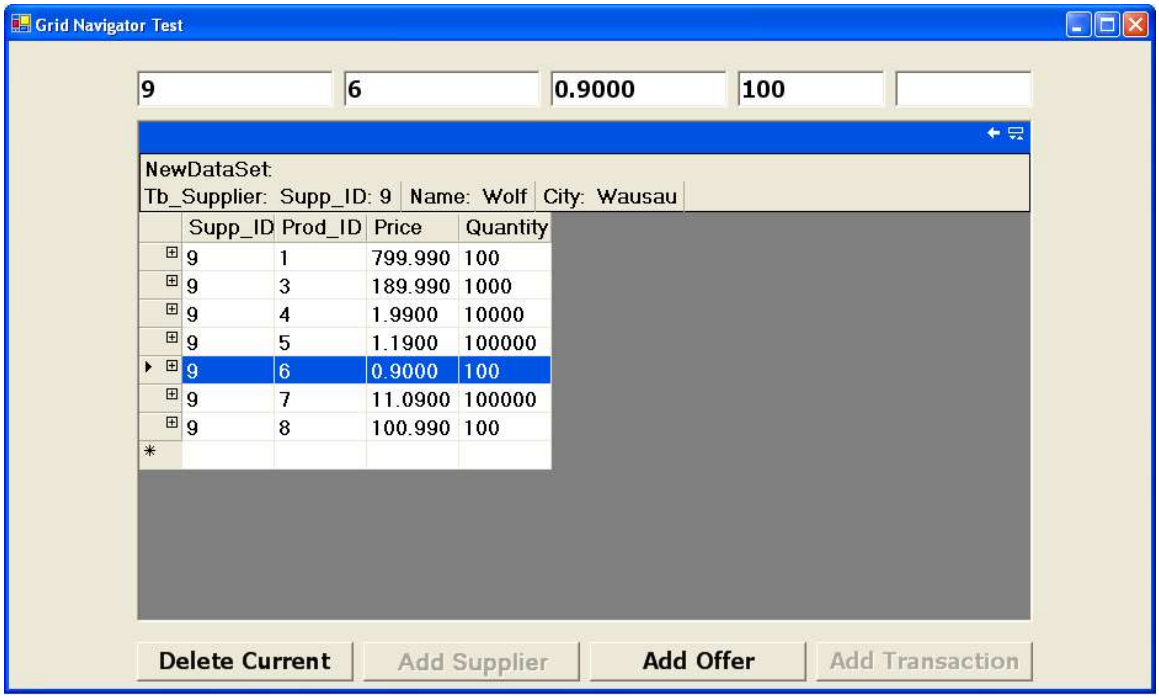

Figure 4: Expected behavior of the DataGrid control.

The latest hypothesis seems to be unlikely because it looks very much like the responsibility of synchronizing the DataGrid with the DataSet is left to the application developer. Although there is no built in functionality to automatically correlate between what we see and what we process in a DataGrid, users can provide their own code to solve this problem by using the existing support provided by the .NET libraries. There are two important pieces that help solving the synchronization problem:

- the GetChildRows() method – featured by all DataRow objects to produce a DataRow array of the dependent rows;

- the DataMember property of the DataGrid object – provides an indication of the table whose rows are visualized and of the path to those rows.

For example a call of the GetChildRows() method from a supplier row object will return the offers of that supplier. Also, given the configuration of the table connections in figure 1, when the grid visualizes rows from the Tb_Transactions table the DataMember property can take one of the following four values:

- "Tb_Transactions" – when the entire content of the Tb_Transactions table is visualized independently;

- "Tb_Supplier.Supplier_Transactions" – when the transaction rows of a selected parent supplier are visualized.

- "Tb_Offers.Offers_Transactions" – when only the transaction rows based on a selected parent offer are visualized, as a result of the child rows of an offer row, by navigating from the Tb_Offers table to the Tb_Transactions table;

- "Tb_Supplier.Supplier_Offers.Offers_Transactions" – when only the transaction rows based on a selected parent offer are visualized, by navigating from the Tb_Supplier table, through Tb_Offers down to the Tb_Transactions table.

By using the GetChildRows() method and the DataMember property one can create the code to properly synchronize a DataGrid with its data source. Let us observe that the CurrentCell.RowNumber property, as introduced in section 2, produces the right synchronization if it is applied to the right subset of rows that is, exactly to the subset of rows visualized in the grid. This subset of rows can be produced by the GetChildRows() method by using the path information in the DataMember property. This assumes keeping track of each navigation step and the corresponding selected parent row. Coding this on an ad-hoc basis can be an option for simple cases, but it can become difficult to control in cases when parent-child navigation goes back and forth across several levels and when the DataSet consists of a larger number of tables with several connections. Therefore, a more comprehensive and application independent approach proves to be beneficial.

4 The GridNavigator

In this section we present the concept and implementation of a GridNavigator as a systematic and generalized approach that solves the problem of correlating the current row in the grid with the corresponding row in the data table. The GridNavigator uses a stack in order to keep track of the navigation paths across the tables thus enabling it to work for arbitrary DataSets and arbitrary configurations of connections between the tables.

4.1 How the GridNavigator Works?

The GridNavigator is supposed to provide proper synchronization both for forward navigation steps and when navigation is backwards. A forward navigation step takes us from a selected parent row to the corresponding detail rows (like the offers of a supplier). Synchronization for such a step is straightforward, since all we have to do is to compute the subset of detail rows by using the GetChildRows() method and apply the CurrentCell.RowNumber property to it. A backward navigation step would take us from a subset of detail rows to their parent row (like from a subset of offers to the supplier making those offers or from the transactions that have been made based on an offer to that offer). A backward navigation step assumes the correct determination of the rows that have been in the grid before the last forward navigation step and determination of the row that has been expanded. There is no direct way of performing such an operation. Therefore the solution we adopted for recreating the status from before the last navigation step is to retrace the entire sequence of the forward navigation steps except for the last

one. There are two pieces of information we need in order to be able to do this. The first one is available in the DataMember property of the DataGrid object. As shown in section 3.1, this gives us the sequence of table links that was followed in the navigation process. The second piece of information we need is the number of the row that was expanded at each forward navigation step. To store this information we use a stack of row number values. A forward navigation step adds a new value to the top of the stack, while a backward step just removes the values from the top of the stack. The entire stack content is scanned from its bottom to its top whenever the DataGrid needs to be synchronized with the underlying data. For each value in the stack we use the corresponding information in the DataMember property of the DataGrid object to find the table link that has been followed at that step and we apply the GetChildRows() method to the row given by the current value in the stack.

4.2 Interface of the GridNavigator Class

The GridNavigator class has a very simple user interface that consists of 3 methods and a constructor.
There are two important events to which a DataGrid control needs to react:

- MouseDown – which is fired when we click in the area of a row in the grid and is used to determine the number of the row that was clicked on;

- Navigate - which is fired when we navigate from one table to the other.

Both of these events require some action in the GridNavigator and are materialized by the two methods that will be called by these events:

- MouseDown – will save in a private local variable the value of the current row number;

- Navigate – will be used to push on the stack the current row number when navigation is forward or to remove the value at the top of the stack when navigating backwards.

Each of these methods is called by simply passing to them the standard parameters of the corresponding event handlers of the DataGrid control.

The third method is GetCurrentDataRow and returns the DataRow object corresponding to the current row in the grid. This method would be called whenever a different row in the grid is selected or when navigating to another table. It takes as parameters the index of the current row in the grid and the value in the DataMember property of the grid control.

The constructor of the GridNavigator would be typically called at form opening and takes as parameter the DataSet to which the DataGrid is connected.

4.3 Implementation of the GridNavigator Class

The code implementing the GridNavigator class is given in the appendix.

The GridNavigator constructor takes as parameter a reference to the DataSet that is, the source of data for the DataGrid control. It creates the stack object used to store the row index values and saves the reference to the dataset in a private variable.

The MouseDown method contains the standard code for determining the index of the current row in the grid and saves it into the currentIndex variable.

The Navigate method updates the stack according to the type of the current navigation step which can be either forward, resulting in pushing a new row index on top of the stack, or backward, in which case the value from the top of the stack is removed (pop operation).

The GetCurrentDataRow method determines the row in the DataSet corresponding to the current index position in the grid. To do this, the stack is scanned from the bottom to its top and the row index values are used at each step to determine the next generation of child rows. In order to be able to explore each value in the stack its content is converted to an array of objects in the navigationArray variable. In parallel the DataMember property of the DataGrid is parsed in order to extract the name of the corresponding table link.

The currentDataMember variable is initialized with the value of the DataMember property of the DataGrid object and will contain at each step the remainder of this string as components of it (either table name or link name) are processed and trimmed out from left to right. This can be nicely controlled by using the `'.'` symbol as delimiter and passing it as parameter to the IndexOf() function. The following line trims the DataMember property string:

```
currentDataMember=currentDataMember.Substring(dotIndex+1);
```

while

```
dotIndex=currentDataMember.IndexOf('.',dotIndex+1);
```

finds the right hand delimiter of the currently processed link.

The conditional expression:

```
dotIndex==-1?
      currentDataMember:
      currentDataMember.Substring(0,dotIndex));
```

returns the name of the table or link to be processed and is given as parameter to the GetChildRows() function computing the set of rows for the next step in the currentTable variable:

```
currentTable=parentRow.GetChildRows(dotIndex==-1?
```

```
                currentDataMember:
                currentDataMember.Substring(0,dotIndex));
```

The ParentRow variable contains at any time the DataRow object that is going to be expanded. Its content is determined by using the row index from the current position in the navigation stack, converted into the navigationArray variable, as an index on the currentTable variable containing the child rows for the current step:

```
parentRow=currentTable[(int)(navigationArray[i])];
```

The final value returned by the GetCurrentDataRow method is a DataRow object as resulted from the successive selections and row expansions recorded in the navigationStack on the path indicated by the DataMember property of the DataGrid.

4.4 Efficiency Tradeoffs – memory/versus speed

As explained in the previous section, our GridNavigator re-computes from scratch the subset of visualized rows for each step on the path of data table links that leads to the rows currently seen in the grid, involving repeated calls of the GetChildRows() method as it can be seen in the implementation of the GetCurrentDataRow method. This may seem quite inefficient from a computational point view for an operation that is performed for every click that changes the current row in the grid. Another approach one may think of would be to store in the stack not only the current row indexes, but also the references to the set of rows produced by the GetChildRows() method at each step. However, this more memory intensive approach is less desirable due to the unpredictability of the amount of data rows that may be stored at times and thus of the amount of memory required by the GridNavigator. This is why the option of re-computing everything from scratch is preferred. It is simple, robust, has a constant memory footprint and proved to be fast enough to provide instant reaction to mouse clicks.

5 Conclusions

The GridNavigator has been developed as a .NET class with a very simple interface and compiled as a DLL ready to be included in applications. In the classroom, to many it comes as a surprise that the DataGrid control does not work entirely as expected and that one need to add our own code in order to fix this. Students developing their projects found the GridNavigator easy to use and felt comfortable including it in various applications.

References

[1] Andrew Troelsen (2003) – "C# and the .NET Platform, Second Edition", *Apress*, 2003.

Appendix

Implementation code of the GridNavigator class.

```
using System;
using System.Collections;
using System.Windows.Forms;
using System.Data;

namespace GridNavigatorNamespace
{
public class GridNavigator
    {
        public GridNavigator(DataSet data)
        {
           this.navigationStack=new Stack();
           this.data=data;
        }
        private Stack navigationStack;
        private int currentIndex;
        private DataSet data;
        public void MouseDown(object sender, MouseEventArgs e)
        {
           DataGrid.HitTestInfo hti=((DataGrid)sender).HitTest(e.X,e.Y);
           if(hti.Type==DataGrid.HitTestType.RowHeader)
               this.currentIndex=hti.Row;
        }

        public void Navigate(NavigateEventArgs ne)
        {
           if(ne.Forward)
                     this.navigationStack.Push(this.currentIndex);
           else
                     this.navigationStack.Pop();
        }

        public DataRow GetCurrentDataRow(int currentGridIndex,
                                                 string gridDataMember)
        {
           string currentDataMember=gridDataMember;
           int dotIndex=gridDataMember.IndexOf('.');
           DataRow[] currentTable=this.data.Tables[dotIndex==-1?
               currentDataMember:
               currentDataMember.Substring(0,dotIndex)].Select();
           DataRow parentRow;
           object[] navigationArray=this.navigationStack.ToArray();
           for(int i=navigationArray.Length-2;i>=0;i--)
           {
               parentRow=currentTable[(int)(navigationArray[i])];
               currentDataMember=currentDataMember.Substring(dotIndex+1);
               dotIndex=currentDataMember.IndexOf('.',dotIndex+1);
               currentTable=parentRow.GetChildRows(dotIndex==-1?
                     currentDataMember:
                     currentDataMember.Substring(0,dotIndex));
           }
           if(currentTable.Length==0) return null;
           return currentTable[currentGridIndex];
        }
    }
}
```