Technical Note

Adapting a Grid to REP++

Author: R&D Department

Publication date: April 28 2006

Adapting a Grid to REP++

Overview

The REP++*toolkit* for Windows® provides a good amount of Windows Forms controls. These controls normally inherit from a .NET counterpart and attach them to the REP++ data and metadata. It is quite common to adapt other Windows Forms controls to REP++. The REP++*toolkit* for Windows already includes a subclass for the .NET Windows Forms **DataGridView** control. This article describes how this adaptation was done.

The DataGridView control

The **DataGridView** control appears in version 2 of .NET. This grid, which supersedes the previous **DataGrid** control, is more powerful and more flexible. To adapt this grid, the following tasks were done:

- Subclassing the **DataGridView** class.
- Binding the grid to a REP++ group instance data.
- Binding the grid to the related REP++ group metadata.
- Validating the user's data.
- Adapting the grid so it can be used in the REP++*framework* for Windows.

To adapt a grid, we must also consider implementing design time support. Design time support provides a way for the grid to be attached to a REP++ source from a list. It also supports a redraw of the grid when its properties are changed.

Subclassing the DataGridView class

The first step is to define a new class that inherits from the **DataGridView** class. The class also implements the **CrdGrpControlUtil.ICrdGrpControl** and **ILineListHandler** interface. The **CrdGrpControlUtil.ICrdGrpControl** interface is used so the class can embed the **CrdGrpControlUtil** object. This object is used to help implement design time support (explained later in the document).

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Windows.Forms;
using System.Windows.Forms.Design;
using RepPP;
using RepPP.Toolkit;

namespace RepPP.Toolkit.Window {
    public class DataGridViewGInst : DataGridView,
                                     CrdGrpControlUtil.ICrdGrpControl,
                                     ILineListHandler {
```

Binding data to the grid

The **DataGridView** control supports the standard Windows Forms data-binding model. The programmer can use a variety of data sources. Alternatively, the programmer can add rows and columns to the control and manually populate it with data (virtual mode). The **DataGridView** subclass in the REP++*toolkit* for Windows adopts the **DataSource** approach. In the future, the subclass may be extended to also use the grid virtual mode to provide the data to it.

The REP++*toolkit* already provides a data source component that connects a control to a group instance (**DataSourceGInst**).

The **DataGridView** grid can easily be directly bound with the **DataSourceGInst** without any additional codes. However, in this implementation, the user of the grid doesn't need to define a **DataSourceGInst** and attach it to the grid. Instead, the grid is embedding its own **DataSourceGInst**. The user can attach the grid at design time with a REP++ card group. At runtime, the **DataGridView** is attached directly to a group instance. To implement this behavior, the class must include a **DataSourceGInst** object and implement some design time support.

The following code embeds a **DataSourceGInst** in the class:

```
private DataSourceGInst                    m_dataSourceGInst;

public DataGridViewGInst() : base() {
    m_dataSourceGInst   = new DataSourceGInst(this, false);
    base.DataSource     = m_dataSourceGInst;
}

protected override void Dispose(bool disposing) {
    if (disposing){
        if (m_dataSourceGInst != null) {
            m_dataSourceGInst.Dispose();
            m_dataSourceGInst = null;
        }
    }
    base.Dispose(disposing);
}
```

Embedding the **DataSourceGInst** is not enough. Some properties that are accessible by the **DataSourceGInst** must now be accessible directly by the grid.

```
[Browsable(false)]
[Description("Data Source attached to a group instance")]
[DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.Hidden)]
protected virtual DataSourceGInst DataSourceGInst {
    get {
        return(m_dataSourceGInst);
    }
}

[DesignerSerializationVisibility(DesignerSerializationVisibility.Hidden)]
[Browsable(false)]
RepPP.Application CrdGrpControlUtil.ICrdGrpControl.Application {
    get {
        return(DataSourceGInst.Application);
    }
}

[DesignerSerializationVisibility(DesignerSerializationVisibility.Visible)]
[Editor(typeof(CrdGrpControlUtil.CrdGrpControlEditor),
              typeof(System.Drawing.Design.UITypeEditor))]
[Browsable(true)]
[Bindable(true)]
[Category("Data")]
```

```
[Description("Name of the CardGroup attached to the source")]
public CrdGrpFullName CardGroupFullName {
    get {
        return(DataSourceGInst.CardGroupFullName);
    }
    set {
        DataSourceGInst.CardGroupFullName = value;
        if (DesignMode) {
            base.DataSource = null;
            base.DataSource = m_dataSourceGInst;
        }
    }
}

[Browsable(false)]
[Description("Group which will be represented by this Data Source")]
[DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.Hidden)]
public virtual Group Group {
    get {
        return(DataSourceGInst.Group);
    }
}


[Browsable(false)]
[Description("GroupInstance represented by this Data Source")]
[DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.Hidden)]
public virtual GroupInstance GroupInstance {
    get {
        return(DataSourceGInst.GroupInstance);
    }
    set {
        DataSourceGInst.GroupInstance = value;
        base.DataSource               = DataSourceGInst;
    }
}


[Browsable(true)]
[Category("Behavior")]
[Description("How to display a choice value")]
public bool ShowDescription {
    get {
        return(DataSourceGInst.ShowDescription ==
               DataSourceGInst.ShowDescriptionE.Always);
    }
    set {
        DataSourceGInst.ShowDescription = (value) ?
                                          DataSourceGInst.ShowDescriptionE.Always :
                                          DataSourceGInst.ShowDescriptionE.Never;
    }
}
```

When the user attaches the grid to a card group, a list of cards and a list of groups in the associated card are proposed. To implement this behavior, some design time support must be implemented. This design time support is already done in part in the **CrdGroupControlUtil** class, and this class is already embedded in the **DataSourceGInst** class. To attach the design time support to the property, the following attribute has been included in its definition:

```
[Editor(typeof(CrdGrpControlUtil.CrdGrpControlEditor),
               typeof(System.Drawing.Design.UITypeEditor))]
```

The class also implements the **CrdGrpControlUtil.ICrdGrpControl** interface. This interface is called by the embedded **CrdGrpControlUtil** object (which is embedded by the **DataSourceGInst** class).

```
void CrdGrpControlUtil.ICrdGrpControl.TriggerComponentChange() {
    DataSourceGInst.TriggerComponentChange();   // Used only in design time.
}

void ControlUtil.IRepControl.RepPPReInit() {
    DataSourceGInst.RepPPReInit();
    // Reinitialize REP++ Metadata. Use only in design time when REP++ is reinitialized.
}
```

The final step to embed the **DataSourceGInst** is to forbid the change of the data source at design time.

```
[Browsable(false)]
[DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.Hidden)]
public new object DataSource {
    get {
        return(DataSourceGInst);
    }
    set {
        // Do nothing
    }
}
```

The property is redefined with the **new** modifier because the base property is not specified as virtual. The property effectively disappears from the designer. Setting the property at runtime also does nothing. However, the **DataGridView** control has a design time interface that permits the change of the **DataSource**. There is not easy way to forbid that.

Binding the grid to the related REP++ group metadata

REP++ provides metadata that can be used to control the grid. If a column of a grid is represented by a combo box, the REP++ metadata can be used to populate its choice list. If the attached REP++ field is not accessible or invisible, the related column can be set as read-only or hidden. The following routine is doing so for a column:

```
protected virtual void SetColumnInfo(DataGridViewColumn col, Field fld,
                                     bool bShowDescription) {
    RepPP.ChoiceList            choiceList;
    string                      strChoice;
    DataGridViewComboBoxColumn  cb;

    if (col is DataGridViewComboBoxColumn) {
        cb = (DataGridViewComboBoxColumn)col;
        choiceList = fld.ChoiceList;
        cb.Items.Clear();
        for (int iIndex = 0; iIndex < choiceList.Count; iIndex++) {
            if (bShowDescription) {
                strChoice = choiceList.GetDesc(iIndex);
                if (strChoice == null || strChoice == String.Empty) {
                    strChoice = choiceList.GetCode(iIndex);
                }
            } else {
                strChoice = choiceList.GetCode(iIndex);
            }
            cb.Items.Add(strChoice);
        }
    } else if (fld.Type == RepPP.FieldType.sdFieldMoney) {
        col.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
    }
    if (col is DataGridViewTextBoxColumn) {
        ((DataGridViewTextBoxColumn)col).MaxInputLength = fld.MaxSize;
    }
    col.ReadOnly = !fld.IsSelectable || fld.IsPrimaryKey;
```

```
    col.Visible  = fld.IsVisible;
}
```

The following routine iterates over the list of columns of the grid and calls the previous routine for each column attached to a REP++ field:

```
protected virtual void SetColumnsInfo() {
    RepPP.Group                     grp;
    RepPP.Field                     fld;
    string                          strPropName;
    bool                            bShowDescription;

    bShowDescription    = ShowDescription;
    grp                 = Group;
    if (grp != null) {
        foreach (DataGridViewColumn col in Columns) {
            strPropName = col.DataPropertyName;
            fld = (strPropName == null) ? null : grp.Fields[col.DataPropertyName];
            if (fld != null) {
                SetColumnInfo(col, fld, bShowDescription);
            }
        }
    }
}
```

This routine has to be called at strategic times:

- When the control is created.
- When a new card group is assigned.
- When REP++ is refreshed.

```
protected override void  OnCreateControl() {
    base.OnCreateControl();
    SetColumnsInfo();
}
```

The following code must be added to these routines:

```
protected override void  OnCreateControl() {
    base.OnCreateControl();
    SetColumnsInfo();
}

[DesignerSerializationVisibility(DesignerSerializationVisibility.Visible)]
[Editor(typeof(CrdGrpControlUtil.CrdGrpControlEditor),
typeof(System.Drawing.Design.UITypeEditor))]
[Browsable(true)]
[Bindable(true)]
[Category("Data")]
[Description("Name of the CardGroup attached to the source")]
public CrdGrpFullName CardGroupFullName {
    get {
        return(DataSourceGInst.CardGroupFullName);
    }
    set {
        DataSourceGInst.CardGroupFullName = value;
        if (DesignMode) {
            base.DataSource = null;
            base.DataSource = m_dataSourceGInst;
        }
        SetColumnsInfo();
    }
}
```

The grid must implement a special behavior for columns attached to a field member of the primary key. The column must be editable if the line is new. Otherwise, the column must be set to read-only. The following code implements this behavior:

```
private int FindColumnIndex(Field fld) {
    int     iRetVal = -1;

    foreach (DataGridViewColumn col in Columns) {
        if (col.DataPropertyName == fld.Name) {
            iRetVal = col.Index;
            break;
        }
    }
    return(iRetVal);
}


private int GetGInstLineFromRowIndex(int iRowIndex){
    GroupInstance       gi;
    DataSourceGInst     dataSourceGInst;
    DataRowGInst        rowGInst;
    int                 iRetVal = -1;

    gi              = GroupInstance;
    dataSourceGInst = DataSourceGInst;
    if (iRowIndex >= 0 && iRowIndex < dataSourceGInst.Count){
        rowGInst = (dataSourceGInst == null) ? null :
                                              dataSourceGInst[iRowIndex] as DataRowGInst;
        if (gi != null && rowGInst != null) {
            iRetVal = gi.LineFromBookmark(rowGInst.Bookmark);
        }
    }
    return(iRetVal);
}


private void SetRowState(int iLineIndex, int iRowIndex){
    DataSourceGInst dataSourceGInst;
    int             iColumnIndex;

    if (GroupInstance.ThisLineState(iLineIndex) == LineState.sdLineNew){
        dataSourceGInst = DataSourceGInst;
        foreach(Field fld in dataSourceGInst.Fields) {
            if (fld.IsPrimaryKey) {
                iColumnIndex = FindColumnIndex(fld);
                if (iColumnIndex != -1) {
                    Rows[iRowIndex].Cells[iColumnIndex].ReadOnly = false;
                }
            }
        }
    }
}

protected override void OnRowEnter(DataGridViewCellEventArgs e) {
    int iNewLine = -1;

    base.OnRowEnter(e);
    iNewLine = GetGInstLineFromRowIndex(e.RowIndex);
    SetRowState(iNewLine, e.RowIndex);
}
```

Validating the user's data

It is very important to be able to validate the data entered by the user. REP++ already provides a good mechanism to validate data. The grid must be able to call the REP++ validation routine at strategic times and to display the error message if necessary.

```
private string                              m_strError;

protected override void OnRowValidating(DataGridViewCellCancelEventArgs e) {
    GroupInstance   gi;
    DataGridViewRow row;
    int             iLineIndex;
    string          strError;
    int             iColIndex;

    base.OnRowValidating(e);
    if (!e.Cancel) {
        gi         = GroupInstance;
        iLineIndex = GetGInstLineFromRowIndex(e.RowIndex);
        if (gi != null && iLineIndex != -1) {
            if (gi.ThisLineState(iLineIndex) != LineState.sdLineDeleted) {
                gi.Group.SaveActiveGroupInstance();
                gi.SaveLinePos();
                try {
                    gi.Select(false, false);
                    gi.SelectLine(iLineIndex, true, false);
                    row             = Rows[e.RowIndex];
                    row.ErrorText   = "";
                    foreach (DataGridViewCell cell in row.Cells) {
                        cell.ErrorText = "";
                    }
                    strError                    = "";
                    gi.Application.FldOnError +=
                              new FldOnErrorEventHandler(Application_FldOnError);
                    foreach (FieldInstance fi in gi.FieldInstances) {
                        m_strError = "";
                        if (fi.Field.Validate() != 0) {
                            e.Cancel    = true;
                            iColIndex   = FindColumnIndex(fi.Field);
                            if (iColIndex != -1) {
                                row.Cells[iColIndex].ErrorText = m_strError;
                            }
                            if (strError != String.Empty) {
                                strError += "\r\n";
                            }
                            strError += fi.Field.Prompt + ": " + m_strError;
                        }
                    }
                    gi.Application.FldOnError -=
                              new FldOnErrorEventHandler(Application FldOnError);
                    if (e.Cancel) {
                        row.ErrorText = strError;
                    } else {
                        if (gi.ValidActiveLine(true) != 0) {
                            e.Cancel = true;
                        }
                    }
                } finally {
                    gi.RestoreLinePos(true);
                    gi.Group.RestoreActiveGroupInstance(true);
                }
            }
        }
    }
}

void Application FldOnError(object sender, FieldGenEventArgs e) {
    m_strError = e.ErrorText;
}
```

The main goal of this routine is to validate each field attached to a cell. If an error is found, the error text is attached to the cell. The error text is also concatenated to a row error list. If any error occurs, the row is flagged as being in error. If no error occurs, the **ValidActiveLine** is called to ensure that all active child data are also valid. One strange thing about this routine

is the necessity to trap the REP++ ***FldOnError*** event. Actually, this is the only way to validate a field and receive the error message.

Adapting the grid so it can be used in the REP++framework for Windows

The REP++*framework* for Windows application is built to accept compatible grids. A grid must implement the **ILineListHandler** interface to be compatible. This interface defines the minimum functionalities a grid must include to be used to display and edit data. The **ILineListHandler** is defined as follows:

```
public interface ILineListHandler {
    event   LineSelectedInListEventHandler LineSelectedInList;
    bool    SelectLine(int iPos);
    bool    InsertLine(bool bAtEnd);
    bool    DuplicateLine();
    bool    DeleteCurrentLine();
    void    CommitEdit();
    void    Refresh();
    void    AssociateGroupInstance(GroupInstance ginst);
    void    OnLineSelectedInList(LineSelectedInListEventArgs e);
}
```

The event is used to inform the framework that a row in the grid is being selected. The **SelectLine**, **InsertLine**, **DuplicateLine**, **DeleteCurrentLine** and **Refresh** methods are used so the framework can do the corresponding function on the grid. **CommitEdit** is called to ensure the edit is done in a row before a validation occurs. Finally, the **AssociateGroupInstance** method associates a new group instance to the grid when a parent line changes.

```
private LineSelectedInListEventArgs         m_evtArgLastRowChange = null;
public event LineSelectedInListEventHandler LineSelectedInList;

protected override void OnSelectionChanged(EventArgs e) {
    base.OnSelectionChanged(e);
    if (m_evtArgLastRowChange != null) {
        OnLineSelectedInList(m_evtArgLastRowChange);
    }
    m_evtArgLastRowChange = null;
}


public bool SelectLine(int iPos) {
    DataGridViewRow row;
    bool            bRetVal;

    if (GroupInstance == null) {
        bRetVal = false;
    } else {
        ClearSelection();
        row = Rows[iPos];
        if (row.Cells.Count > 0) {
            CurrentCell = row.Cells[0];
            row.Selected = true;
        }
        bRetVal = true;
    }
    return(bRetVal);
}

public bool InsertLine(bool bAtEnd) {
    DataGridViewRow row;
    bool            bRetVal;
```

```
    if (GroupInstance == null) {
        bRetVal = false;
    } else {
        if (!AllowUserToAddRows) {
            DataSourceGInst.AddNew();
            row = Rows[Rows.Count - 1];
            if (row.Cells.Count > 0) {
                CurrentCell = row.Cells[0];
                CurrentCell.Selected = true;
            }
        }
        bRetVal = true;
    }
    return(bRetVal);
}

public bool DuplicateLine() {
    bool            bRetVal;
    DataGridViewRow row;

    if (GroupInstance == null) {
        bRetVal = false;
    } else {
        if (CurrentRow == null) {
            bRetVal = false;
        } else {
            DataSourceGInst.AddDuplicate();
            row = Rows[Rows.Count - 1];
            if (row.Cells.Count > 0) {
                CurrentCell = row.Cells[0];
                CurrentCell.Selected = true;
            }
            bRetVal             = true;
        }
    }
    return(bRetVal);
}


public bool DeleteCurrentLine() {
    bool    bRetVal;

    if (GroupInstance == null) {
        bRetVal = false;
    } else {
        if (CurrentRow != null) {
            Rows.Remove(CurrentRow);
            bRetVal = true;
        } else {
            bRetVal = false;
        }
    }
    return(bRetVal);
}

public void CommitEdit() {
    CommitEdit(DataGridViewDataErrorContexts.Commit);
}

void ILineListHandler.Refresh() {
    GroupInstance   ginst;

    ginst         = GroupInstance;
    GroupInstance = null;
    GroupInstance = ginst;
}

public void AssociateGroupInstance(GroupInstance ginst) {
    GroupInstance = ginst;
}
```

```
public void OnLineSelectedInList(LineSelectedInListEventArgs e) {
    if (LineSelectedInList != null) {
        LineSelectedInList(this, e);
    }
}
```

Finally, the following routine needs to be changed as follow:

```
protected override void OnRowEnter(DataGridViewCellEventArgs e) {
    int iOldLine = -1;
    int iNewLine = -1;

    base.OnRowEnter(e);
    if (CurrentRow != null) {
        iOldLine = GetGInstLineFromRowIndex(CurrentRow.Index);
    }
    iNewLine = GetGInstLineFromRowIndex(e.RowIndex);
    m_evtArgLastRowChange = new LineSelectedInListEventArgs(iOldLine, iNewLine);
    SetRowState(iNewLine, e.RowIndex);
}
```

The full source of the **DataGridViewGInst** class can be found in the *RepPP.Toolkit.V1* project.