Designing and implementing an customizable table view for various data structures using OOP with C++

Ing. Dan Andrei Stanciu
Universitatea "Tibiscus" Timişoara

REZUMAT. O vizualizare tabelară a datelor definite de o aplicație este foarte utilă pentru a crea o imagine de ansamblu dintr-o singură privire asupra conținutului. Pentru aceasta s-a dezvoltat cu ajutorul programării orientate pe obiecte o componentă grafică aptă de a reprezenta date pe rânduri şi coloane, total controlabilă din punct de vedere al implementării şi care oferă multe posibilități de manevrare şi particularizări de comportament şi aspect grafic.

Tibiscus

This paper will debate on the advantages of OOP component design and will exemplify them though the design and implementation of a table or grid control. A grid control that is data-aware, that has a connection with a list of data, thus creating a framework similar to the MFC document/view framework, the list of data representing the document that is viewed through the grid control, but also this framework is best used by interfacing it with the MFC document/view framework.

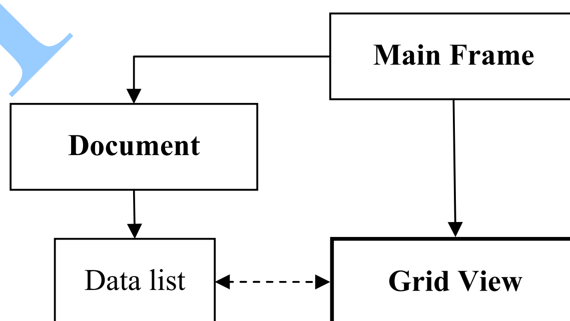

Figure 1 ***The interaction within the MFC framework and the data-grid framework***

The drawing above can be extrapolated by having multiple MainFrame objects, multiple Document objects and of course, multiple Data Lists and Grid Views. It should also be specified that there can be only one Grid View and multiple Data Lists that will be bound successively to the Grid View object.

The structural OOP design

A grid representation must consist of two parts that interact: the grid header, that contains the columns of the grid, usually directly related to the fields in the data list displayed and the grid body, which is the container for this data.

The grid header should contain one or more columns, should control the resizing and moving of these columns. Each column has a caption that will be displayed in its client area. Also, a column should be able to display icons in its client area whenever the data in the grid is ordered by this column.

The grid body displays the data that is bound to it, distributing every field in a data record to the corresponding place in the grid. A necessary feature of the grid body is to handle the selection mechanism, so the user can interact with one or more data rows at a time using the mouse or the keyboard. Also, the grid body has the main role in handling scrolling events if the number of rows and/or columns overrides the client area of the grid body.

The grid body contains a finite number of rows. At this level the abstraction of components starts. The grids rows can embrace various forms. The most obvious would be the abstract model of a row containing cells, according to the columns in the grid header. Another abstraction of a grid row will be the grid band, which represents a special grid row and has a specific role in the grid body.

This abstract model of the grid rows offers multiple possibilities for deriving different kinds of grid rows, each for different usage, and integrating them into the grid body.

The class design

The class `CGridView` is derived from the standard CView MFC class, which completes the MFC document/view architecture by adding the ability to visualize the document represented by a CDocument object, the main part of the document/view architecture.

Tibiscus

This `CGridView` object is a container for the two components specified above: the grid header (class `CGridHeader`) and the grid body. The header contains the list of columns (class `CGridColumn`). The grid body owns the list of rows (the abstract definition of a row – class `CGridRow`), and each row that can contain cells (`CGridCell`) must be an instance of `CGridCellRow`. Here is the class diagram that can make things easier to understand:

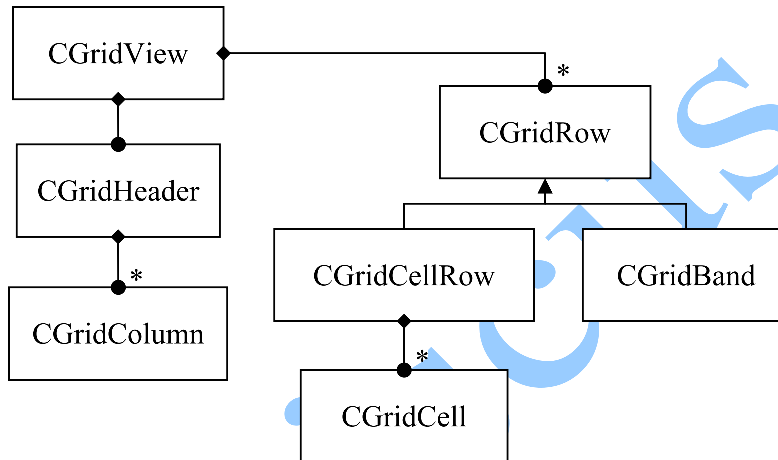

***Figure 2** Grid class diagram*

Here is one of the skins of the grid control, and the visual representation of these classes in the grid client area:

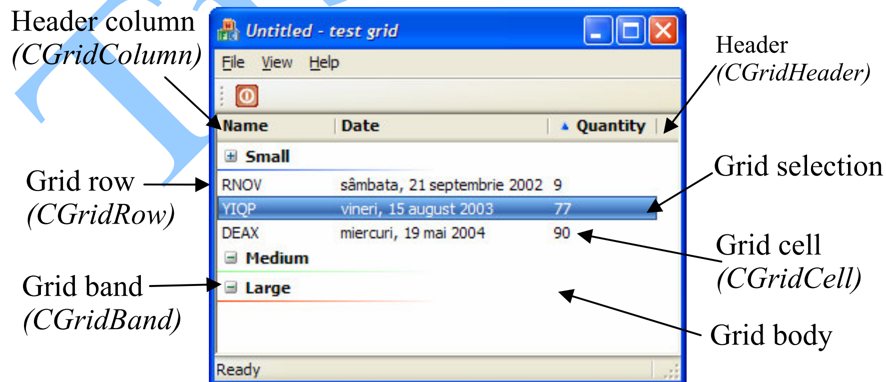

***Figure 3** The visual representation of the grid control*

Developing

Drawing

One of the most important aspects in deriving objects with the purpose of creation of new controls is the painting. In the grid component presented here, the painting aspect is relevant not only for the visual representation of the grid, for its design, but also for illustrating some advanced object oriented programming techniques.

In the Microsoft Visual C++ integrated development environment (IDE), the painting surface of any visual object is called a *Device Context*. This abstract notion is implemented by the MFC class `CDC`. The CDC class offers a group of primitive functions that can be used for rendering shapes on a painting surface.

Normally, any function that is meant to render an object's client area has the following form:

```
void Draw(CDC *pDC);
```

The drawing of the grid control implies the graphical rendering of each of its components. Thus, each of these components must paint itself and paint its subcomponents if any. For example, the `CGridView` class will paint the header and the body area. Thus, the Draw method will look like this:

```
void CGridView::Draw(CDC *pDC)
{
// render grid background with <white>
    pDC->FillSolidRect(CRect(0,GetHeader().GetHeight(),
                     m_nWidth,m_nHeight),
RGB(255,255,255));
// render grid body – invoke each row's Draw() method
    for (int k=0; k<Rows.GetCount(); k++)
          if (Rows[k]->IsShowing())
                Rows[k]->Draw(pDC);
// render the grid header – invoke the header's Draw()
methd
    m_pHeader->Draw(pDC, nLeftX);
}
```

So the `CGridView` class is clearing the client area (filling it with white), then drawing the rows and then rendering the header. Also, the rows that contain cells will need to paint each of them – invoke each cell's `Draw()` method.

As you can notice in *Figure 2*, the `CGridRow` class is the base class for `CGridRowCell` and `CGridBand`. The `CGridRow` is an abstract class; its

Lucius

purpose is to offer a starting point for deriving custom objects that can be used as rows in the grid. Both CGridCellRow and CGridBand represent rows in the grid, the difference is the first can contain cells and the latter cannot, its functionality is totally different. The similarity is that both classes are abstract representations of rows in the grid.

An abstract class is a class that doesn't implement all of its methods. Methods of a class can be left unimplemented by declaring them as *pure virtual functions*. Pure virtual functions are functions that have no implementation in the parent class; their implementation is to be found in one of the subclasses of this abstract class. Here is how the CGridRow class is declared:

```
class CGridRow : public CObject
{
   ...  // other implemented methods

   virtual void Draw(CDC *pDC) = 0;
   virtual BOOL IsShowing() const = 0;
   ...
}
```

So, the Draw() function in class CGridRow is declared as a pure virtual function, that has no implementation (pure virtual functions can be spotted by the particular " = 0" after their declaration). Also, the IsShowing() constant method (the method is declared with the const modifier because it doesn't modify the contents of the class) is declared as pure virtual; this function can be used for checking if the row will be showed when the grid will be rendered (as you can see in the code sequence for the grid body rendering).

Then, the CGridCellRow class has the following declaration/implementation:

```
// in GridCellRow.h
class CGridCellRow : public CGridRow
{
   ...
   CTypedPtrArray <CPtrArray, CGridCell*> Cells;

   virtual void Draw(CDC *pDC);
   BOOL IsShowing() const;
}

// in GridCellRow.cpp
void CGridCellRow::Draw(CDC *pDC)
{
```

```
    ...  // other rendering
    for (int k=0; k<Cells.GetCount(); k++)
         Cells[k]->Draw(pDC);
}

BOOL CGridCellRow::IsShowing() const
{
    return TRUE;
}
```

The `CGridBand` will also implement both `Draw()` and `IsShowing()` methods. The `Draw()` from the `CGridBand` class method will render the caption of the band and some gradient decoration as an underline or as a background.

Data

In the beginning of this paper I mentioned that the grid control is data-aware. This means there is a bilateral connection established between the rows of the grid and the records in the dataset. Every row has a unique correspondent in the dataset, like the following drawing is showing:

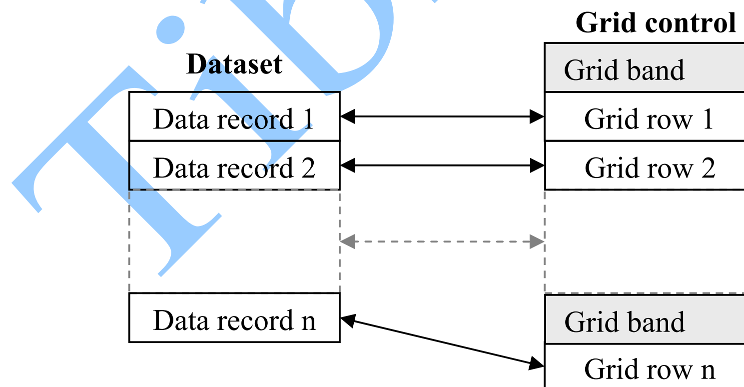

Figure 4 *Dataset and grid control correspondence*

The basic classes for the dataset framework are the `CData` and the `CDataItem` classes. The `CDataItem` represents a single record in the dataset, while the `CData` class is the abstraction for the dataset object itself. Both of these classes are abstract, they need to be subclassed to be instantiated and

Tibiscus

used in a project. The `CDataItem` declares the pure virtual function `GetOutputValue()` like this:

```
virtual CString GetOutputValue(UINT nCol) const = 0;
```

The purpose of this function is to return the string representation of the data column identified by the `nCol` parameter, in the implementation of the subclass. This way, each member field in the dataset that needs to be represented in the grid will have a string representation. For example, the integer value `12` will have to be converted into string, thus obtaining the value `"12"`; a date/time field having a value of `26.04.2005, 16:47:03` could be shown in the grid as "`2005, the 26th of April, at 16:47`".

The `CData` class is also abstract, by declaring the `CreateItem()` pure virtual function. Here is a piece of the header file of the `CData` class:

```
class CData : public CObject
{
protected:
    UINT m_nFieldCount;

    CTypedPtrList <CPtrList, CDataItem*> Items;

public:
    virtual CDataItem* CreateItem() = 0;
    ...
}
```

The `CreateItem()` function should return an instance of a subclass of the `CDataItem` class (because the `CDataItem` class cannot be instantiated) like this:

```
class CPersonalDataItem : public CDataItem {
public:
    CString m_strName;
    BOOL m_bMarried;
    int m_nChildrenCount;
    CTime m_dBirthday;

    ... // other functions
};

class CPersonalData : public CData {
public:
    CPersonalData() : CData() {
         m_nFieldCount = 4;
    }
```

```
    CDataItem* CreateItem() {
         return new CPersonalDataItem(this);
    }
};
```

The relation between a grid row and a data record is marked in the implementation of the application by the insertion of items in the grid. The grid control defines the `InsertRow()` member function for inserting data rows, and also the `InsertBand()` function for adding bands:

```
void InsertRow(CDataItem* pDataItem,
                  int nBandID = -1, int index = -1);
void InsertBand(UINT ID, LPCSTR pszCaption = "",
                  COLORREF color = GRIDCTRL_COLOR_BAND);
```

As you can see, every row inserted will have its corresponding data item (this will usually be a subclass of the `CDataItem` class). Each row can be defined as belonging to a certain group in the grid. Groups (class `CGridBand`) are identified by ID numbers. This connection between rows and their associated groups is made (not only) when inserting rows in the grid, through the `nBandID` parameter.

The rows of the grid are declared in the grid as an array of abstract grid rows that can be data rows or bands, like this:

```
CTypedPtrArray <CPtrArray, CGridCtrlRow*> Rows;
```

Reference

[GHJV01] **Gamma, E., Helm, R., Johnson, R., Vlissides, J**. *Design Patterns – Elements of Reusable Object-Oriented Software,* Addison Wesley, 2001

[JK00] **Jamsa, K., Klander, L.** (2000), *Totul despre C şi C++ Manualul fundamental de programare în C şi C++*, Teora, 2000

[Sto00] **Stoicu–Tivadar, V**. (2000) *Programare orientată pe obiecte*, Orizonturi universitare, 2000

[Wil98] **Williams, M.** (1998) *Bazele Visual C++4*, Teora, 1998

Tibiscus