

**Figure 18** Key Strip Control Property Configurations

Property	Description
AllowItemReorder	Like with Office 2003 toolbars, users can reposition strip items by setting this property to True. Also like with Office 2003, the user needs to press Alt before repositioning. (Excluding ContextMenuStrip.)
CanOverflow	When the form becomes narrower or shorter than a strip control, this option specifies how to visually represent tool strip items hidden by the overlap; either hidden if False or accessible via a dropdown at the end of the strip control if True.
Dock	Specifies the form edge to line up against or fill.
GripStyle	Specifies whether a MenuStrip or ToolStrip displays a grip, allowing it to be repositioned.
Renderer	Sets a custom ToolStripRenderer implementation to provide non-default tool strip UIs. Causes RenderMode to be set to Custom.
RenderMode	Strip controls support four render modes. System: rendered with colors from the system palette. Professional: rendered according to the current Windows Color Scheme, being either Blue, Olive Green, or Silver. ManagerRenderMode: rendered using the Windows Theme sensitive ToolStripRenderer—and is the default. Custom: can't be set explicitly—see Renderer property.
ShowItemToolTips	Sets whether strip items can also display tooltips. (False for MenuStrip and ContextMenuStrip.)
SizingGrip	Sets whether to hide or show the form sizing grip. (StatusStrip only.)

Strip control items can be added, updated, and deleted via the Properties window, with smart tags, or visually from the design surface. Both MenuStrip and ToolStrip provide an “Insert Standard Items” option to pre-populate the controls with items for typical commands such as Open, Save, Save As, Cut, Copy, and Paste.

The strip control Properties window and smart tag support enable a variety of other interesting configurations, the most important of which are controlled by the properties listed in **Figure 18**. Apart from the standard properties you would typically find on controls, such as Enabled, Visible, Checked, and BorderStyle, strip controls offer some additional properties, outlined in **Figure 19**.

## Strip Container

One of the key features of the strip control suite is the ability to drag them from one edge of a form to another.

**Figure 19** Key Tool Strip Item Control Properties

Property	Description
Alignment	Determines which end of a tool strip control a tool strip item control will align to, either Left or Right.
DisplayStyle	Specifies what elements to display, including None, Text, Image, ImageAndText.
ImageScaling	Specifies how an image is rendered, either ShrinkToFit or None.
Overflow	Determines how each strip item control responds to overflow. Can be either Never, Always, or AsNeeded, which leaves the decision up to the host strip control.
TextImageRelation	Sets the position of text in relation to an image. Applies when DisplayStyle is ImageAndText.

While strip controls are docked to form edges by default, docking doesn't provide the smarts to support dragging. Instead, this behavior is added to a form with a ToolStripContainer control, shown in **Figure 20**.

Because ToolStripContainer is a container control that provides form-wide strip control semantics, it needs to be added to a form before any other controls. Controls are then added to its content panel, while strip controls are added to one of four special panels.

Each panel can be expanded or collapsed to accommodate the strip controls you'd like against each form edge. You can also hide/show panels as needed by setting the ToolStripContainer's TopToolStripPanelVisible, RightToolStripPanelVisible, BottomToolStripPanelVisible, and LeftToolStripPanelVisible Boolean properties.

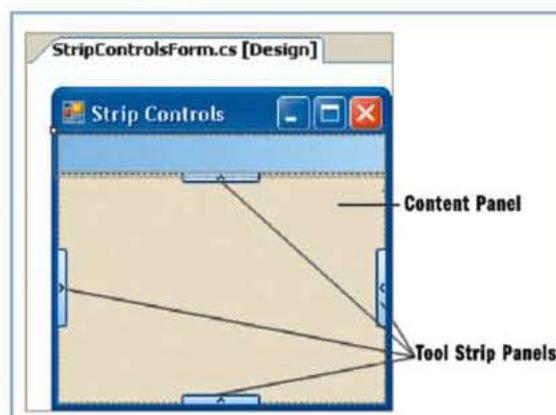
At run time, you can drag strip controls between the edges of the form for which panels are visible. Only strip controls that have a drag grip can be dragged. And if two or more tool strip controls reside in the same tool strip panel, they can be repositioned amongst

**Strip control items can be added, updated, and deleted via the Properties window, with smart tags, or visually from the design surface.**

themselves as required by the user. Since tool strips can be moved to new positions by users at run time, users will expect them to be in those positions from one application session to the next. For this, you use the ToolStripManager class (from System.Windows.Forms), which nicely implements the static (shared) LoadSettings and SaveSettings methods.

## Data Binding

In Windows Forms 2.0, data binding is much richer than before thanks

**Figure 20** ToolStripContainer Control



**Figure 21** Data Sources

to the new **BindingSource** component whose primary role is to upgrade non-data binding-savvy data sources with richer data-binding capabilities. This enables the unification of disparate data sources and, in addition, facilitates the creation of a single client code pattern for any type of data source.

A data source can be easily added to a project using the Data Source Configuration Wizard (Data | Add New Data Source). From there, you can create data sources for databases, Web services, and objects. If you create a data source from a database, a new strongly typed **DataSet** is added to your project and can be viewed from the new Data Sources window (Tools | Data | Show Data Sources), shown in **Figure 21**.

The Data Sources window provides tool strip items that allow you to add new data sources and, if a data source is a strongly typed **DataSet**, edit, reconfigure, or refresh them. When dropped onto a form, controls like **DataGridView** will automatically prompt you to select a data source or create a new one.

## BindingSource

When you think “data source,” you are likely to think of the most popular type: the typed **DataSet**. But data sources come in many shapes and sizes, including tables, XML, and .NET-based objects. All of these could be bound to in Windows Forms 1.x, with differing levels of data binding integration. This forced developers to spend a nontrivial amount of time updating their data sources to achieve higher levels of data binding integration by implementing **IBindingList**. The Windows Forms 2.0 **BindingSource** component addresses this problem by being able to consume any list type and reexposing these as **IBindingList** list data source implementations. Additionally, item types can be handed to the **BindingSource** component, which turns them into an **IBindingList** implementation for you.

To configure a **BindingSource** to consume the desired data source, drop one onto your form and set its **DataSource** and **DataMember** properties appropriately. Here’s an example of **BindingSource** consuming a typed **DataSet**:

```
public partial class DataBindingForm : Form {
    public DataBindingForm() {
        ...
        // Consume a type data set
        this.bindingSource.DataSource = this.northwindDataSet;
        this.bindingSource.DataMember = "Employees";
    }
}
```

The **BindingSource** component now exposes a strongly typed **DataSet** as a data source against which controls can bind.

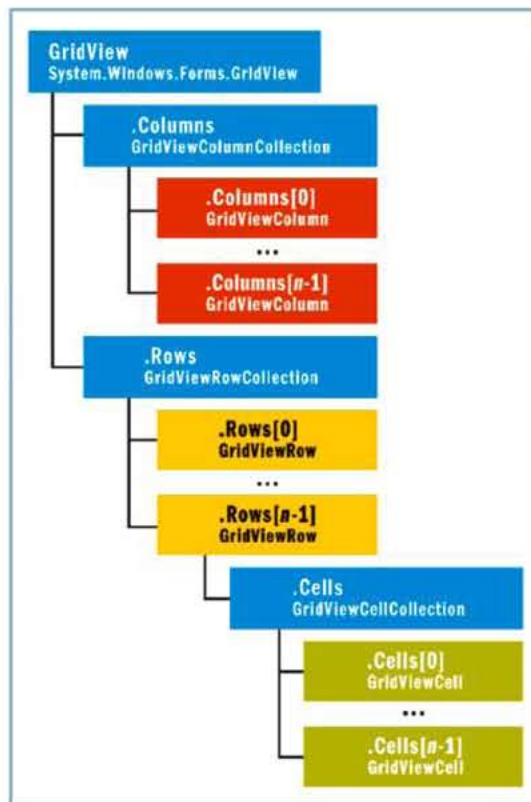
## DataGridView

The Windows Forms team took the opportunity to respond to community feedback on the **DataGrid** by constructing a wholly new grid control, **System.Windows.Forms.DataGridView**. (**DataGrid** still exists for navigating and editing hierarchical data.)

One striking difference between **DataGrid** and **DataGridView** is the latter’s object model, which has been abstracted into a natural grid structure composed from columns, rows, and cells, illustrated by **Figure 22**. This natural grid structure allows developers to quickly and logically drill down into a ton of intuitively located functionality, including:

- Rich UI customization support through styles, formatting, layout, and item selection
- The ability to natively display a broader variety of data types than the **DataGrid**, including images
- Nifty features like frozen columns and run-time column reordering
- More than 100 events for granular control over navigation, editing, validation, custom painting, and error handling

One of the new capabilities is row selection on cell click, which is configured via the **DataGridView**’s **SelectionMode** property. Of course, no control can be everything to everyone. When they have to, developers can lean on extensibility to incorporate custom features. The **DataGridView** infrastructure offers a variety of cell, row and column base implementations you can derive from and plug in, including **DataGridViewColumn** and **DataGridViewCell**. All this makes the **DataGridView** a much more compelling grid-style control than the **DataGrid**.



**Figure 22** **DataGridView** Object Model

The most common way to fill a

**DataGridView** is to bind it to a data source, which can be achieved by setting its **DataSource** property to a **BindingSource**, as shown in **Figure 23**.

While **DataGridView** displays multiple items, you sometimes want to view one item at a time. In this case, you can just as easily bind controls like the **TextBox** to a **BindingSource**. But when you view one item at a time, you need a way for users to navigate between items. Such support is typically provided by a VCR-style control. Rather than creating your own, you can drag a **BindingNavigator** onto a form and bind it to the same **BindingSource** your controls are bound to. **BindingNavigator** also provides current item and total item count information, as well as support for both adding and removing items to and from the data source.

