RadioPanel: Binding RadioButton Groups to Enumeration Properties

Jay Andrew Allen

28 Aug 2007 CPOL

Provides a Windows Forms control that enables binding the value of an enumeration type property to a Panel of RadioButtons.

Download source - 17.57 KB

Introduction

This article introduces a control that supports a simplified type of data binding between a panel of `RadioButton` controls and an enumerated property.

Background

On the .NET Windows Forms projects I've worked on, the piddling problem that's vexed me more than any other is the lack of support for `RadioButton` handling. Sure, you can stick a bunch of `RadioButton`s in a `Panel` or `GroupBox`, and Windows Forms will kindly enforce a minimal grouping behavior on the set, enabling only one radio to be selected at any one time. The problem arises when you need to know which `RadioButton` was checked. Neither `Panel` nor `GroupBox` surface an event or property that will tell you this; it's left to you to provide an event handler for every single `RadioButton` in the group. Needless to say, with this poor support, databinding easily to `RadioButton` group is naught but a pipe dream.

Other people much smarter than I have already proffered up solutions to this problem. The best is probably Duncan MacKenzie's RadioButtonList control, which emulates the corresponding ASP.NET control. While good, this solution struck me as severe overkill. What I wanted was a simpler solution that used standard radio buttons - a custom `Panel` control that provided two-way data binding to an `enum`-typed property on a business object, as well as a single event notification indicating when the current selection has changed. I didn't need a "no-code" approach, necessarily - just an approach that greatly simplified the spaghetti code that inevitably accompanies `RadioButton` handling.

My solution, `RadioPanel`, requires minimal wiring: a handful of designer settings, plus a single line of code. In an application requiring two, three or more `RadioButton` panels, this spells a sizable reduction in code.

Using the Code

Figure 1 below shows the simple business object represent a `Vessel`. Our `Vessel` has a `Type` property signified by the `VesselType` enumeration:

Hide Copy Code

```
public enum VesselType
{
    Cargo = 0,
    Container,
    BulkCarrier,
    Reefer,
    Passenger,
    Tanker
}
```

We can use `RadioPanel` to create a new `Panel` on our form containing one button for each of these options. To pull off our binding, we leverage the fact that every `Control` in Windows Forms 2.0 exposes a `Tag` property that accepts arbitrary data. We'll use the `Tag` property on each `RadioButton` by populating it with the numeric value of its corresponding option in our `VesselType` enumeration. Since "`Cargo`" is the first option in `VesselType`, we assign `rbVesselTypeCargo.Tag` a value of `0`. We then set `rbVesselTypeContainer.Tag =1`, and so on for all of our `RadioButton` options.

`RadioPanel` supports a property named `ValueMember`, indicating the property we wish to bind to. We give this a value of "`Type`." Finally, inside of our form's `Load` event, we create an instance of our business object, and assign it to the `DataSource` property of `RadioPanel`:

Hide Copy Code

```
private Vessel v;

public Form1()
{
    InitializeComponent();

    ...

    v = new Vessel();
    v.Type = VesselType.Passenger;
    vesselButtonPanel.DataSource = v;
}
```

You can run this application, and use the debugger to verify that the `Vessel.Type` property is updated whenever you select a new vessel type. You can use the "Set Value" button on the form to verify that the binding goes both ways: when the `Vessel` object is updated, so is the `RadioButton` selection.

Points of Interest

The code driving this makes fairly straightforward use of .NET's powerful `Reflection` API. Reflection enables you to address properties, methods and events on objects at run-time. This is known as late-binding, as opposed to the early binding performed during compilation: instead of specifying your property and method calls statically in code, you can obtain a generic reference to an object and invoke its properties and methods using `string` parameters.

`RadioPanel` kicks off the proceedings once it detects that you have assigned it valid `ValueMember` and `DataSource` properties. The `DataSource` property uses `Object.GetType()` to get a reference to the data source's underlying type. It then employs the `Type.GetInterface()` method to discover whether our data source supports `INotifyPropertyChanged`. If so, it uses `Type.GetEvent()` to get a handle to the `PropertyChanged` event, and sets up a listener.

Hide Copy Code

```
Type iface = _dataSource.GetType().GetInterface("INotifyPropertyChanged");

if (iface != null)
{
    _ei = iface.GetEvent("PropertyChanged");

    if (_ei != null) // which would be weird if it did...
    {
        _ei.AddEventHandler(_dataSource, _pceh);
    }
}
```

The code for obtaining the value of the `ValueMember` property proceeds in a similar fashion. The `Type.GetProperty()` method returns an object of type `PropertyInfo`, which supports reading and setting the value of the underlying property using the `GetValue()` and `SetValue()` methods, respectively. This code is fairly straightforward, so I won't reproduce it here; see the full project for the details.

How do we know when a `RadioButton` selection has changed? Simple: we listen for the `CheckedChanged` event of each `RadioButton`, using the `ControlAdded` method of our `Panel` parent as our entry point for the event wiring:

Hide Copy Code

```
protected override void OnControlAdded(ControlEventArgs e)
{
    base.OnControlAdded(e);

    if (e.Control is RadioButton)
    {
        RadioButton rb = (RadioButton)e.Control;
        rb.CheckedChanged += rb_CheckedChanged;
    }
}
```

`rb_CheckedChanged` then takes care of firing the `RadioButtonSelectionChanged` event, as well as setting the newly selected value on our `DataSource`. Note that we use a boolean flag to prevent processing the `PropertyChanged` notification that this Reflection code is going to generate. Perhaps the most "magical" part of this code is the call to `Enum.Parse()`,which converts the integer discovered in `RadioButton.Tag` to an instance of the `VesselType` enumeration.

Hide Shrink Copy Code

```
void rb_CheckedChanged(object sender, EventArgs e)
{
    if (_dataSource != null)
    {
        int nSetting = 0;

        if (sender is RadioButton)
        {
            RadioButton rbSender = (RadioButton)sender;

            // Fire the RadioButtonChanged event.
            FireRadioSelectionChanged(rbSender);

            if (rbSender.Tag == null)
            {
                throw new InvalidOperationException("RadioButton " + rbSender.Name +
                 " does not have its Tag property set to a valid enum integer value.");
            }

            if (!Int32.TryParse(rbSender.Tag.ToString(), out nSetting))
            {
                throw new InvalidOperationException("RadioButton " + rbSender.Name +
```

```
                    " does not have its Tag property set to a valid enum integer value.");
            }

            PropertyInfo pi =
        (PropertyInfo)_dataSource.GetType().GetProperty(_valueMember);
            if (pi != null)
            {
                // Convert the int into its corresponding enum.
               // pi.PropertyType represents the enum's type.
                object parsedEnum;
                try
                {
                    parsedEnum = Enum.Parse(pi.PropertyType, nSetting.ToString());
                }
                catch (Exception ex)
                {
                    throw new InvalidOperationException
               ("Could not convert RadioButton.Tag value into an enum.", ex);
                }

                // Stop listening to property changes while we change the property
               //  - otherwise, stack overflow.
                _processPropertyChange = false;

                pi.SetValue(_dataSource, parsedEnum, null);

                _processPropertyChange = true;
            }
        }
    }
}
```

For most purposes, however, you won't have to worry about all of this "under the curtain" wiring; just add the control to your control library, and use it to your heart's content. I think you'll find that it tidies up your Windows Forms interface code quite nicely.

History

- 28th August, 2007: Initial post

License

Share

About the Author

Jay Andrew Allen

Web Developer

United States

Jay Andrew Allen is a software developer with over a decade of experience. He spent over six years at Microsoft Corporation, where he helped add new features to Windows Forms 2.0, assisted Premier developer customers in debugging n-tier Web applications, and wrote core documentation for the .NET Framework. He currently works in C# and Visual Basic.NET building Windows Forms and Windows P...

show more

Comments and Discussions

You must Sign In to use this message board.

Search Comments

First Prev Next

Awesome!
Member 12641569 19-Jul-16 10:14

Awesome, thanx
HennieSmit 21-Aug-12 23:40

My vote of 5
Iakovos Karakizas 2-Aug-12 2:17

My vote of 5
igetorix 27-Jul-11 11:46

Excellent control & a suggestion
Arkitec 25-May-11 10:05

Very Helpful
Ahmed Shawky Zidan 4-Mar-11 1:20

Thanks for the example.
edge9421 20-Nov-10 16:37

Thanks
trantrum 11-Aug-09 11:25

Thanks!

Sitary **3-Sep-07 12:52**

Re: Thanks!
Jay Andrew Allen 3-Sep-07 12:58

Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.