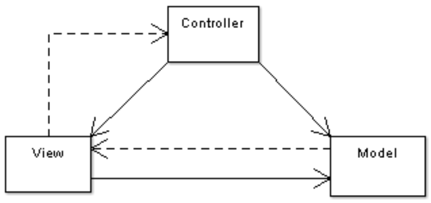
Model-View-Controller for a Windows.Forms Application

Wiktor Zychla [wzychla_at_ii.uni.wroc.pl]

Download the MVCDemo source code here (22kb)

Introduction

Model-View-Controller (MVC) is an **architectural** pattern that separates application's data model and user interface views into separate components. This is what the definition says. However, to fully understand the MVC we have to introduce required terms and point benefits of MVC.



A wiki image of MVC

The **data model** is a set of data structures that lay the base for the businnes logic of an application. In typical object-oriented application, the data model is built of client-side classes and collections. The data model typically is somehow stored into a Database Management System, however how the data is exactly stored is not a concern of MVC.

The **view** is a user interface element (typically a window) that presents the data to the user and allows the user to change the data. It is a typical situation to have several views active at the same time (for example in the Multiple-Document Interface).

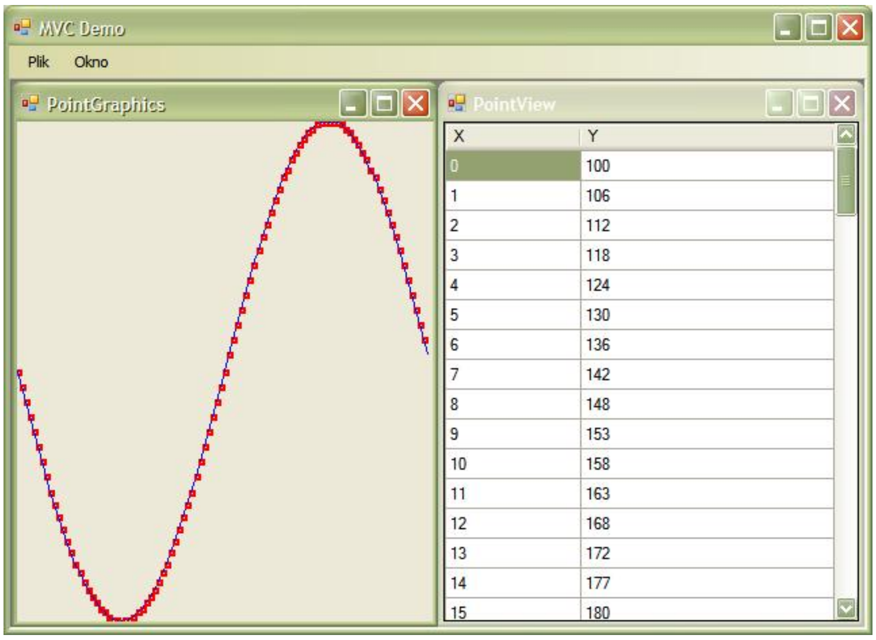
The **controller** is a connection point between the model and views: views register in the controller and reports changes in the model to them.

Having model and views separated seems correct from the perspective of object-oriented design but at the same time we face a serious problem: **how do views communicate?** or more precisely **how the changes made to the model should be at once reported to all views?**

MVC gives you the the answer to this question.

Example

Our example application will let the user to play with functions. We store 100 function arguments and values and we let the user to see the data in a grid and in a graphical chart. Both types of views must be interactive.



X	Y
0	100
1	106
2	112
3	118
4	124
5	130
6	136
7	142
8	148
9	153
10	158
11	163
12	168
13	172
14	177
15	180

Model

We start by building the model. Since we are working on points, let's start by defining a point class.

```
public class Point
{
    private int x;
    public int X
    {
        get { return x; }
        set { x = value; }
    }

    private int y;
    public int Y
    {
        get { return y; }
        set { y = value; }
    }

    private Point() { }
    public Point( int x, int y )
    {
        this.x = x;
        this.y = y;
    }
}
```

Now we need a model with storage for points inside. Node that the internal dynamic collection is not directly exposed. Instead, we expose it as a more restrictive type of collection.

```
public class Model
{
    private List<Point> points;
    public Point[] Points
    {
        set
        {
            points = new List<Point>( value );
        }
        get
        {
            return points.ToArray();
        }
    }

    public void AddPoint( int y )
    {
        points.Add( new Point( points.Count, y ) );
    }

    public Model()
    {
        points = new List<Point>();
    }

    public void Initialize( int Points, int MaxValue )
    {
        for ( int p = 0; p < Points; p++ )
            AddPoint( (int)( MaxValue / 2 + Math.Sin( 2 * Math.PI * p / Points ) * MaxValue / 2 ) );
    }
}
```

We also need a default instance for the model and here we can use a singleton pattern.

```
public class Model
{
    #region Singleton
    static Model instance;
    public static Model Instance
    {
        get
        {
            if ( instance == null )
                instance = new Model();

            return instance;
        }
    }
    #endregion

    ...
}
```

View

A view must able to react to changes in the model and/or raise changes in the model caused by user/external interaction.

We start by defining how changes in the model will be exposed. Since our model focuses on points, we only need to inform anyone that one of points is changed. However, in real-life applications, when the model is much more complicated, the amount of details you put into the notification is limited only by your imagination.

```
public class ModelChangeEventArgs
{
    public int PointIndex;

    private ModelChangeEventArgs() {}
    public ModelChangeEventArgs( int PointIndex )
    {
        this.PointIndex = PointIndex;
    }
}
```

Views are described in an uniform way

```
public interface IView
{
    void ModelChange( object sender, ModelChangeEventArgs e );
}
```

and to be the view, a form must only implement the interface

```
public partial class PointGraphics : Form, IView
{
  ...
```

What should happen when the notification about the model beeing changed is received by a view? Well, it depends on the view itself and the notification. In our example the nofitication says: "hello, one of points is changed" and the grid view will update only one cell

```
public partial class PointView : Form, IView
{
    public PointView()
    {
        InitializeComponent();

        pointBindingSource.DataSource = Model.Instance.Points;
    }

    #region IView Members

    public void ModelChange( object sender, ModelChangeEventArgs e )
    {
        if ( e.PointIndex >= 0 && e.PointIndex < Consts.POINTS )
        {
            DataGridViewCell cell = dataGridView1.Rows[e.PointIndex].Cells[1];

            cell.Value = Model.Instance.Points[e.PointIndex].Y;
            dataGridView1.InvalidateCell( cell );
        }
    }

    #endregion
```

while the graphical view will be just completely rebuilt:

```
public partial class PointGraphics : Form, IView
{
    public PointGraphics()
    {
        InitializeComponent();

        this.SetStyle( ControlStyles.AllPaintingInWmPaint | ControlStyles.OptimizedDoubleBuffer | ControlStyles.UserPaint, true );
    }

    private void PointGraphics_Paint( object sender, PaintEventArgs e )
    {
        double xstep = (double)this.ClientRectangle.Width / Model.Instance.Points.Length;
        double ystep = (double)this.ClientRectangle.Height / Consts.MAXH;

        for ( int p = 0; p < Model.Instance.Points.Length - 1; p++ )
        {
            using ( Pen pn = new Pen( Color.Red, 2 ) )
                e.Graphics.DrawRectangle( pn, new Rectangle( (int)( p * xstep ), (int)( Model.Instance.Points[p].Y * ystep ), 3, 3 ) );
            using ( Pen pn = new Pen( Color.Blue, 1 ) )
                e.Graphics.DrawLine( pn,
                    new PointF( (int)( p * xstep ), (int)( Model.Instance.Points[p].Y * ystep ) ),
                    new PointF( (int)( ( p + 1 ) * xstep ), (int)( Model.Instance.Points[p + 1].Y * ystep ) ) );
        }
    }

    #region IView Members

    public void ModelChange( object sender, ModelChangeEventArgs e )
    {
        Refresh();
    }

    #endregion
```

Controller

The controller is a central part of MVC. Active views register here and the controller is able to raise the notification of model changes.

```
public class Controller
{
    #region Singleton
    private static Controller instance;
    public static Controller Instance
    {
        get
        {
            if ( instance == null ) instance = new Controller();

            return instance;
        }
    }
    #endregion

    public delegate void ModelChangeDelegate( object sender, ModelChangeEventArgs e );
    public event ModelChangeDelegate ModelChangeEvent;

    public void RegisterView( IView view )
    {
        this.ModelChangeEvent += new ModelChangeDelegate( view.ModelChange );
    }
    public void UnregisterView( IView view )
    {
        this.ModelChangeEvent -= new ModelChangeDelegate( view.ModelChange );
    }

    public void RaiseModelChange( object sender, ModelChangeEventArgs e )
    {
        if ( ModelChangeEvent != null )
            ModelChangeEvent( sender, e );
    }
}
```

Let them play together

We have the model, we have views and we have the controller. We only need two more things:

- views have to register/unregister in the controller
- views have to raise the model change event when the user changes the data

As for the first issue, we register views when they are created and unregister when they are disposed:

```
private void PointView_Load( object sender, EventArgs e )
{
    Controller.Instance.RegisterView( this );
}

private void PointView_FormClosed( object sender, FormClosedEventArgs e )
{
    Controller.Instance.UnregisterView( this );
}
```

As for the second issue, in cause of any user interaction raises the model change event. In a grid view we monitor changes in the DataGridView:

```
private void dataGridView1_CellValueChanged( object sender, DataGridViewCellEventArgs e )
{
    Controller.Instance.RaiseModelChange( this, new ModelChangeEventArgs( e.RowIndex ) );
}
```

In a graphical view we allow the user to select points with mouse and move them vertically:

```
private void PointGraphics_MouseMove( object sender, MouseEventArgs e )
{
    if ( MovedPoint != null )
    {
        MovedPoint.Y += (int)( ( e.Y - LastY ) / ( (double)ClientRectangle.Height / Consts.MAXH ) );
        LastY = e.Y;

        Controller.Instance.RaiseModelChange( this, new ModelChangeEventArgs( MovedPoint.X ) );
    }
}
```

Conclusion

MVC focuses on clean separation of data model and data views. Multiple views simultaneously respond to changes in data but in the same time are rather independend of each other. As the number of different but concurrent views increase, the complexity of handling mutual changes does not change at all. Since the above example could be easily generalized, I think it could even be provided as a standard mechanism of future versions of System.Windows.Forms.