

Model View Presenter

Decoupling and separation, while no new concept, has become more and more of a mantra for me. One of the patterns I have come to consider most valuable, if not completely indispensable, is “Model View Presenter” (MVP for short). I find that it adds considerably to any application with a graphical user interface, so in this my first post I will introduce an implementation I use, and find simple and powerful, yet extensible.

Model View Presenter basically moves the logic for managing how a user interface behaves and how it interacts with the core of an application, away from the interface itself. By decoupling the control logic from a specific implementation like Windows Forms or WPF, the user interface becomes easily interchangeable and the code that runs it becomes much more manageable.

MVP consists of three main components, two of which will be my primary focus in the following. The components are: the model, the view and the presenter. The view is the graphical representation shown to the user. The model is the “actual” application. The presenter contains the logic for controlling how the view behaved. Displaying data, retrieving input and triggering functionality provided by the model.

In the following I will focus mainly on the “outer most” components, the view and the presenter. In a later post I hope to touch on techniques for building the much more complex model / domain.

The following example will be in C# and I will use Windows Forms for the concrete interface implementation. C# is my .Net “language of choice” and Windows Forms is widely used and understood (although the implementation wouldn’t be much different in WPF).

Let’s look at some code.

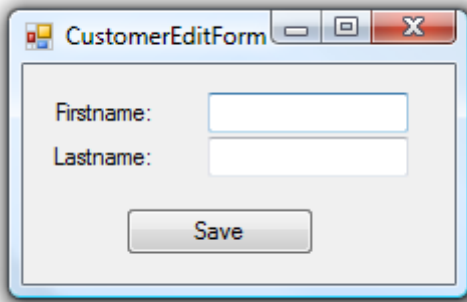
The View

```
public interface ICustomerEditView
{
    string FirstName { get; set; }
    string LastName { get; set; }

    event EventHandler Save;
}
```

The above interface defines a view for editing a customer. The system is going to display the first and last name of a customer, so the interface must contain two properties for accepting those (setters). The purpose of the view is to allow the user to change the first and last name, so options to retrieve the values are needed as well (getters). Finally the view needs the ability to send a signal when the user decides to save changes. That’s what the “Save” event is for.

A concrete implementation of the view described above might be something as complicated as what is seen below.



Here's a quick look at the code for the form above.

```
public partial class CustomerEditForm : Form, ICustomerEditView
{
    public Form1()
    {
        InitializeComponent();
    }

    #region ICustomerEditView Members

    public string FirstName
    {
        get { return txtFirstName.Text; }
        set { txtFirstName.Text = value; }
    }

    public string LastName
    {
        get { return txtLastName.Text; }
        set { txtLastName.Text = value; }
    }

    public event EventHandler Save;

    #endregion

    private void btnSave_Click(object sender, EventArgs e)
    {
        if (this.Save != null)
            this.Save(sender, e);
    }
}
```

The properties defined by the ICustomerEditView interface forward their values to the Text properties of the corresponding TextBox controls. In the event handler for the Save button, we just fire the Save event (after checking that somebody is actually listening).

The Model

For the model part of my example I have defined a simple interface with two methods for loading and saving a customer, and a property that holds an id identifying the selected customer (set on some form we

have no knowledge of – maybe not exactly the way it would be done in the real world, but I’m sure you get the point).

```
public interface Model
{
    int SelectedCustomerId { get; }
    Customer LoadCustomer(int id);
    void SaveCustomer(int id, string FirstName, string LastName);
}
```

The Presenter

```
public class CustomerEditPresenter
{
    private ICustomerEditView m_view;
    private IModel m_model;

    public CustomerEditPresenter(ICustomerEditView view, IModel model)
    {
        m_view = view;
        m_model = model;

        Initialize();
        WireViewEvents();
    }

    private void Initialize()
    {
        Customer customer = m_model.LoadCustomer(m_model.SelectedCustomerId);
        m_view.FirstName = customer.FirstName;
        m_view.LastName = customer.LastName;
    }

    private void WireViewEvents()
    {
        m_view.Save += new EventHandler(m_view_Save);
    }

    void m_view_Save(object sender, EventArgs e)
    {
        if (m_view.FirstName != "" && m_view.LastName != "")
            m_model.SaveCustomer(m_model.SelectedCustomerId, m_view.FirstName,
m_view.LastName);
    }
}
```

I’ll go thru the presenter step by step. First off there is two private members to hold references to the model and view dependencies. The presenter will do its work by reacting to events, accessing properties and calling methods on these components.

```
private ICustomerEditView m_view;
private IModel m_model;
```

The view and model is injected thru the constructor (*constructor injection*) and stored in the private members. Next are calls to two private setup methods WireViewEvents and Initialize. This is just what I like

to call them, but they could just as well be one method or the code could be written directly in the constructor.

```
public CustomerEditPresenter(ICustomerEditView view, IModel model)
{
    m_view = view;
    m_model = model;

    Initialize();
    WireViewEvents();
}
```

Let's start with Initialize. This is where the view is set up for its initial presentation. The selected customer is loaded from the model and the "FirstName" and "LastName" properties of the Customer is assigned to the corresponding properties on the view. The view is now ready to be displayed (we'll do that later – don't worry).

```
private void Initialize()
{
    Customer customer = m_model.LoadCustomer(m_model.SelectedCustomerId);
    m_view.FirstName = customer.FirstName;
    m_view.LastName = customer.LastName;
}
```

WireViewEvents takes care of hooking up eventhandlers on the view. In this case the only event is the Save event.

```
private void WireViewEvents()
{
    m_view.Save += new EventHandler(m_view_Save);
}
```

The view eventhandler contains the actual code that handles the saving of changes to the customer. First we make sure that both first- and lastname have values (else we simply do nothing. In the real world we might tell the view to display some sort of error message to the user). If validation is successful the model is called and the customer is saved.

```
void m_view_Save(object sender, EventArgs e)
{
    if (m_view.FirstName != "" && m_view.LastName != "")
        m_model.SaveCustomer(m_model.SelectedCustomerId, m_view.FirstName,
m_view.LastName);
}
```

All we need to do now is actually display the view to the user. For the purpose of demonstration this is done in program.cs in the Main method.

```
ICustomerEditView view = new CustomerEditForm();
IModel model = new Model() { SelectedCustomerId = 5 };
CustomerEditPresenter customerEditPresenter = new CustomerEditPresenter(view,
model);

Application.Run((Form) view);
```

I will go thru this line by line. First a new instance of the concrete view type CustomerEditForm is created. This is the actual Windows Forms form we saw earlier.

```
ICustomerEditView view = new CustomerEditForm();
```

Then a new instance of the concrete model type is created, and the "SelectedCustomerId" property is set to 5.

```
IModel model = new Model() { SelectedCustomerId = 5 };
```

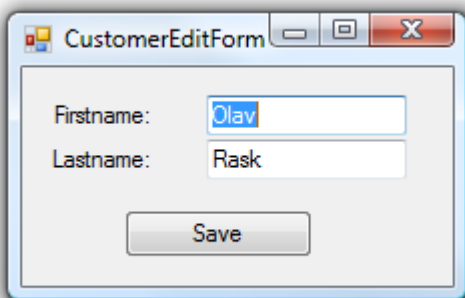
Now the presenter can be created, passing the two dependencies - the view and the model – as parameters.

```
CustomerEditPresenter customerEditPresenter = new CustomerEditPresenter(view, model);
```

And finally we pass the view (cast as a form), to the Application.Run() method.

```
Application.Run((Form) view);
```

The form should now show up looking something like this.



Finally when the Save button is clicked, the dummy implementation of IModel.SaveCustomer, will print out the customer's id, followed by the values of "FirstName" and "LastName", in the output window.

And there you have it: Model View Presenter.

Final thoughts

As discussed earlier, one purpose of MVP is to abstract the actual implementation of the user interface from the code running it. This is achieved very efficiently by using event to signal the presenter. In projects I have worked on I have set up a structure where the only assembly referenced by the user interface is a common interface library defining the view interfaces. By loading the user interface via reflection, there is now no longer any need for assemblies referencing the user interface.

I have found the technique described in this article extremely useful and would not consider starting a project containing a graphical interface without it. I implement the communication between the view and the presenter in pretty much exactly this way, but there are lots of subjects not covered here, that will

make this pattern even more effective. Workflow is one of the first things that come to mind. How will the application switch between forms? Also the implementation of a model containing state, employed here, is not an approach I would recommend – especially not for larger projects.

I aim to touch on these subjects in future posts. Until then I hope somebody takes something away from this short article, and finally any feedback will be much appreciated!

The source for the example is available from my blog mrrask.wordpress.com.

- Olav Rask, 2008 – mrrask.wordpress.com.