Presenter injection in Model-View-Presenter pattern with StructureMap

Asked 15 years, 10 months ago Modified 15 years, 2 months ago Viewed 4k times

6

I've implemented my own copy of the model view presenter pattern (in vein of web client software factory) so I can leverage my own DI framework instead of being tied to WCSF's ObjectBuilder which I had numerous problems with. I've come up with a few ways to do it but none of them particularly make me happy. I wanted to know if anyone else had some other ideas.

Solution #1a

Uses a HttpModule to intercept context.PreRequestHandlerExecute to call ObjectFactory.BuildUp(HttpContext.Current.Handler)

```
public partial class _Default : Page, IEmployeeView
{
    private EmployeePresenter _presenter;

    private EmployeePresenter Presenter
    {
        set
        {
            _presenter = value;
            _presenter.View = this;
        }
    }
}
```

Solution #1b

Call buildup in page load instead of using a HttpModule

```
public partial class _Default : Page, IEmployeeView
{
    private EmployeePresenter _presenter;

    private EmployeePresenter Presenter
    {
        set
        {
            _presenter = value;
            _presenter.View = this;
        }
    }
```

```
    protected void Page_Load(object sender, EventArgs e)
    {
        ObjectFactory.BuildUp(this);
    }
}
```

Solution #1c

Access presenter through Property allow Getter to BuildUp if needed.

```
public partial class _Default : Page, IEmployeeView
{
    private EmployeePresenter _presenter;

    public EmployeePresenter Presenter
    {
        get
        {
            if (_presenter == null)
            {
                ObjectFactory.BuildUp(this);
            }

            return _presenter;
        }
        set
        {
            _presenter = value;
            _presenter.View = this;
        }
    }
}
```

Solution #2

```
public partial class _Default : Page, IEmployeeView
{
    private EmployeePresenter _presenter;

    private EmployeePresenter Presenter
    {
        get
        {
            if (_presenter == null)
            {
                _presenter = ObjectFactory.GetInstance<EmployeePresenter>();
                _presenter.View = this;
            }

            return _presenter;
        }
    }
}
```

Solution #2b

```
public partial class _Default : Page, IEmployeeView
{
    private EmployeePresenter _presenter;

    private EmployeePresenter Presenter
    {
        get
        {
            if (_presenter == null)
            {
                Presenter = ObjectFactory.GetInstance<EmployeePresenter>();
            }

            return _presenter;
        }
        set
        {
            _presenter = value;
            _presenter.View = this;
        }
    }
}
```

Edit: Added solution 1c, 2b

c# dependency-injection structuremap mvp

Share
Improve this question
Follow

edited Feb 6, 2009 at 18:24

asked Feb 5, 2009 at 5:02

Chris Marisic
33.1k ● 30 ● 168 ● 261

4 Answers

Sorted by: Highest score (default)

7

+50

I'd Use solution #1b, and create a layer supertype for all the pages, in order to DRY the presenter initialization a bit more. like this:

Page code:

```
public partial class _Default : AbstractPage, IEmployeeView
{
    private EmployeePresenter presenter;

    private EmployeePresenter Presenter
    {
        set
        {
```

```
            presenter = value;
            presenter.View = this;
        }
    }
    protected override void Do_Load(object sender, EventArgs args)
    {
        //do "on load" stuff
    }

}
```

Abstract Page code:

```
public abstract class AbstractPage : Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        ObjectFactory.BuildUp(this);
        this.Do_Load(sender,e);
        //template method, to enable subclasses to mimic "Page_load" event

    }
    //Default Implementation (do nothing)
    protected virtual void Do_Load(object sender, EventArgs e){}
}
```

With this solution you have the presenter initialization (created by the ObjectFactory) in one class only, if you need to modify it later you can do it easily.

Edit:

Should Do_Load be *abstract* or *virtual* ?

Template Method originally states that the method should be Abstract, in order to force subclasses to implement it, adhering to the superclass contract. (see wikipedia example of "Monopoly" < "Game").

On the other hand in this particular case, we don't want to force the user class to redefine our method, but give it the chance to do so. If you declare it abstract, many classes will be obliged to redefine the method just to leave it empty (this is clearly a code smell). So we provide a sensible default (do nothing) and make the method virtual.

Share

Improve this answer

Follow

edited Feb 13, 2009 at 17:48

answered Feb 10, 2009 at 5:38

Pablo Fernandez

105k ● 58 ● 195 ● 233

This will probably be the accepted answer the only thing I'm having problems on deciding is whether Do_Load is better marked virtual or abstract. – Chris Marisic Feb 12, 2009 at 15:58

1

I've built my own MVP framework like that too. I found the best way for me was to use generics with a base page class. By specifying the Presenter type in the generic class definition, I get to miss out most of the code which each of your proposals requires.

However, there are some things I do not like about doing it that way. The class definition can end up looking rather complicated, and isn't easy to read for a newbie. I also haven't completely worked out a good way to use the event model in the base page.

Sorry I don't have the code for you here, but I can post some for you if you wish. I also have an old version of the code posted at www.codeplex.com/aspnetmvp, should you want to see how it works.

Share Improve this answer Follow

answered Feb 14, 2009 at 9:04

1

I have been using a base page class with:

```
protected override void OnInit(EventArgs e)
    {
        StructureMap.ObjectFactory.BuildUp(this);
        base.OnInit(e);
    }
```

The base class approach works on user controls as well, that alone kept me from the module (didn't want to have 2 ways to set it up). For the page it is

```
public partial class Employee : View, IEmployeeView
{
    public ViewPresenter Presenter { get; set; }
    private void Page_Load(object sender, EventArgs e){}
}
```

I inject the view through the constructor. To avoid the circular reference issue on the structuremap config, just use this helper method:

```
static T GetView<T>()
{
    return (T) HttpContext.Current.Handler;
}
```

On the structuremap config use a convention for both the presenter and the view injection.

Share

Improve this answer

Follow

edited Oct 9, 2009 at 18:00

answered Feb 14, 2009 at 8:02

eglasius
36k ● 5 ● 68 ● 110

0

Thank you all for your very valuable input. Your answers each gave me valuable ideas to combine together in my final solution and this is what I came up with:

```
public abstract class ViewBasePage<TPresenter, TView> :
    Page where TPresenter : Presenter<TView>
{
    protected TPresenter _presenter;

    public TPresenter Presenter
    {
        set
        {
            _presenter = value;
            _presenter.View = GetView();
        }
    }

    /// <summary>
    /// Gets the view. This will get the page during the ASP.NET
    /// life cycle where the physical page inherits the view
    /// </summary>
    /// <returns></returns>
    private static TView GetView()
    {
        return (TView) HttpContext.Current.Handler;
    }

    protected override void OnPreInit(EventArgs e)
    {
        ObjectFactory.BuildUp(this);
        base.OnPreInit(e);
    }
}
```

And inherited by my original page:

```
public partial class _Default :
    ViewBasePage<EmployeePresenter, IEmployeeView>, IEmployeeView
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!Page.IsPostBack)
        {
            _presenter.OnViewInitialized();
        }

        _presenter.OnViewLoaded();
        Page.DataBind();
    }
```

```
    #region Implementation of IEmployeeView

    ...

    #endregion
}
```

Share Improve this answer Follow

answered Feb 15, 2009 at 21:41

Chris Marisic

33.1k ● 30 ● 168 ● 261