A Method for Portable Business Logic: *Model-View-Presenter*

Bowen, Patrick (BHGE); 2019/05/21

Usage

Use the Model-View-Presenter (MVP) approach to ensure separation of concerns, portability of business logic, enable hot-swapping of Views and Models, enable insulation of components from breaking changes, enable easy study of contract between view and business logic.

Best suited for programs with one or few Views.

Contents

Structure

Actors

- A **View** such as the user or hardware interface
- A **Presenter** – a thin bridge between View/s and Model/s
- A **Model** containing business logic

Interfaces

- IPresenter is empty; you could instead forbid concrete **Presenter** public members/methods.
- IView has methods updating **View**, and events passing data/requests to **Presenter**.
- IModel has methods updating **Model**, and events passing data/requests to **Presenter**.

Initialisation

View should initialise and retain an instance of IPresenter, passing only itself as an IView through the **Presenter's** constructor. (Or, consider initialising the view and presenter separately in Program.cs).

Presenter should retain the IView, and initialise and retain instance/s of IModel/s.

Rules and Advice

- Mitigate **View** dependency on **Presenter** or **Model** namespaces
- Forbid **Presenter** dependency on **View** namespaces
- Forbid **Model** dependency on **View** or **Presenter** namespaces
- Mitigate use of non-native .NET types for passing data/requests
- Forbid value-return methods in IView and IModel

MVP Project Checklist

From the description of MVP earlier in this document, we can derive the following requirements:

1. There is an IModel interface.
    a. Has events for **Model** → **Presenter** interaction.
    b. Has methods for **Presenter** → **Model** interaction.
2. There is an IView interface.
    a. Has events for **View** → **Presenter** interaction.
    b. Has methods for **Presenter** → **View** interaction.
3. There is an IPresenter interface.
    a. It is empty.
4. There is at least one **Model** concrete class.
    a. It implements IModel.
5. There is at least one **View** concrete class.
    a. It implements IView.
    b. There is an IPresenter private field.
    c. Its constructor is at least `public View () => _presenter = new Presenter(this);`
6. There is at least one **Presenter** concrete class.
    a. It implements IPresenter.
    b. There is an IView private field.
    c. There is an IModel private field.
    d. Its constructor is at least `public Presenter (IView view) => _view = view;`
7. *Rules and Advice* is strictly followed.

Example Console Project – “Doubler”

This example project solely has business logic to double a number but could do any number of actions and data returns. The code is marked with (x.x.) for items of the *MVP Project Checklist*.

Program.cs

Initialises the program. In this case, we want to initialise a **View** – in WinForms, it would be a Form.

```
static class Program
{
    static void Main (string[] args) => new ViewDoubler();
}
```

IModelDoubler.cs

Defines a ‘contract’ between the **Model** and the **Presenter**.

```
public interface IModelDoubler
{
    event Action<int> NumberDoubled;
    void DoubleNumber (int number);
}
```

1.

1.a.

1.b.

IViewDoubler.cs

Defines a ‘contract’ between the **Presenter** and concrete **View**.

```
public interface IViewDoubler
{
    event Action<int> NumberForDoubling;
    void DisplayDoubledNumber (int number);
}
```

2.

2.a.

2.b.

IPresenter.cs

Forbids the **View** to interact with its instance of the **Presenter**.

```
public interface IPresenter { }
```

3.a.

ModelDoubler.cs

Implements IModel, with concrete business logic.

```
public class ModelDoubler : IModelDoubler
{
    public event Action<int> NumberDoubled; //Implements IModelDoubler
    public void DoubleNumber (int number)
        => NumberDoubled?.Invoke(number * 2); //Implements IModelDoubler
}
```

4.a.

ViewDoubler.cs

Implements the IView, with concrete logic concerned only with this particular UI.

```
public class ViewDoubler : IViewDoubler             5.a.
{
    public event Action<int> NumberForDoubling; //Implements IViewDoubler

    private readonly IPresenter _presenter;       5.b.

    public ViewDoubler ()
    {
        _presenter = new PresenterDoubler(this);        5.c.
        BeginUserPrompt();
    }

    private void BeginUserPrompt ()
    {
        Console.Write("Enter number to be doubled: ");
        var number = int.Parse(Console.ReadLine() ?? "0");
        NumberForDoubling?.Invoke(number);
    }

    public void DisplayDoubledNumber (int number) //Implements IViewDoubler
        => Console.WriteLine($"Doubled is: {number}");
}
```

PresenterDoubler.cs

Implements IPresenterDoubler, with concrete logic to prepare data for the **View**.

```
public class PresenterDoubler : IPresenter        6.a.
{
    private readonly IViewDoubler _view;          6.b.
    private readonly IModelDoubler _model;        6.c.

    public PresenterDoubler (IViewDoubler view)
    {
        _view = view;                           6.d.
        _model = new ModelDoubler();

        _view.NumberForDoubling += _model.DoubleNumber;

        _model.NumberDoubled += _view.DisplayDoubledNumber;
    }
}
```

Rationale

For example, implementing a service with many commands into, and much data out of, a business model. Imagine having to implement its **View** as a WinForm, Webpage, Console app, Android app and Tcp server simultaneously. Eliminating as much business logic as possible from the UI means more code reuse, and less rewriting.

Also imagine that the IModel/**Model** changes, but the Views are already very well built. The Presenter can insulate the View from these changes.