



The Model-View-Controller(MVC) Pattern with C#/WinForms



Volynsky Alex

30 Dec 2013 CPOL

Briefly and clearly about MVC implementation, without lengthy discussion or minor details

[Download demo - 10.5 KB](#)

[Download source - 33.6 KB](#)

Introduction

This article is used to demonstrate the MVC Pattern in .NET using C#/WinForm.

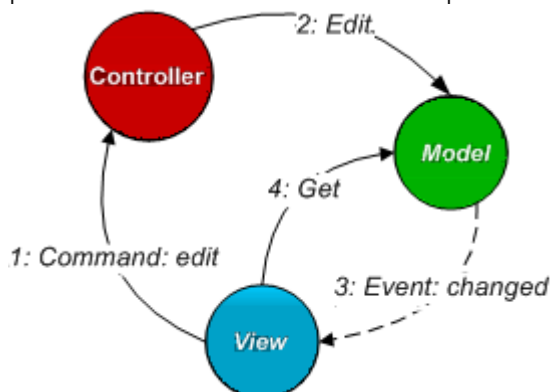
Here a simple "User Information Manager" application which is organized according the the Model-View-Controller(MVC) pattern.

The application displays a list of contacts and allows you to add, modify, and delete existing contacts. The idea is to separate the user interface into View (creates the display, calling the Model as necessary to get information) and Controller (responds to user requests, interacting with both the View and Controller as necessary). The main advantage of MVC pattern is Loose Coupling. All the layers are separated with their own functionality. It is easy to replace a layer with some other type of layer. In other words, MVC pattern is to break up UI behavior into separate pieces in order to increase reuse possibilities and testability. I am using Visual Studio 2010 Ultimate with .NET 4.0 for creating this Application.

Background

Model-View-Controller as the name applies considers three pieces:

- **Model:** it should be responsible for the data of the application domain
- **View:** it presents the display of the model in the user interface
- **Controller:** it is really the heart of the MVC, the intermediary that ties the Model and the View together, i.e. it takes user input, manipulates the model & causes the view to update




For more information about MVC, please see the following [article from Wikipedia](#)

The Solution

The User Information Manager is an application where you can store your customers' contact information. The application displays a list of contacts and allows you to add, modify, and delete existing contacts. All customers have an ID, first name, last name and sex. The screen that operator of this app uses to maintain his list of customers could look something like this:

Id	First Name	Last Name	Department	Sex
122	Vladimir	Putin	Government of Russia	Male
123	Barack	Obama	Government of USA	Male
124	Stephen	Harper	Government of Canada	Male
125	Jean	Charest	Government of Quebec	Male
126	David	Cameron	Government of United Kingdom	Male
127	Angela	Merkel	Government of Germany	Female
128	Nikolas	Sarkozy	Government of France	Male
129	Silvio	Berlusconi	Government of Italy	Male
130	Yoshihiko	Noda	Government of Japan	Male

List of customers can be viewed, added, removed and updated (currently it contains only V.I.P Members, but if you want to become a member of the club, just ask  Smile | `` and no problems, it's free). After a new user is added, his ID cannot change anymore.

The Class Diagram

In the design of a system, a number of classes are identified and grouped together in a class diagram which helps to determine the relations between objects.

The Description of Components

Part of Controller

In order to detach logic from the View, we have to make a View feel as helpless as possible, so we'd prefer to make the Controller do all the hard work and just hand the View some simple commands that do not require any further processing. According to our design, we do this by defining an interface, `IUsersView`, which the View must implement. This interface contain only the signatures of properties/methods we need to use.

```
using System;
using WinFormMVC.Model;

namespace WinFormMVC.Controller
{
    public interface IUsersView
    {
        void SetController(UsersController controller);
        void ClearGrid();
        void AddUserToGrid(User user);
        void UpdateGridWithChangedUser(User user);
        void RemoveUserFromGrid(User user);
    }
}
```

```

string GetIdOfSelectedUserInGrid();
void SetSelectedUserInGrid(User user);

string FirstName    { get; set; }
string LastName     { get; set; }
string ID           { get; set; }
string Department   { get; set; }
User.SexOfPerson Sex { get; set; }
bool CanModifyID    {      set; }
    }
}

```

Now we have a fairly good interface with number of methods. Even if the MVC pattern formally declares that the Controller should receive the events and act upon the View, is often more practical and easier to have the View subscribe to the events and then delegate the handling to the Controller.

Finally I show the actual realization of the Controller (see the **UsersController** class). He hooks up the Model (**User** class) with View (**UserView** class).

```

public class UsersController
{
    //Notice we only use the interfaces. This makes the test more
    //robust to changes in the system.
    IUsersView _view;
    IList      _users;
    User       _selectedUser;

    //The UsersController depends on abstractions(interfaces).
    //It's easier than ever to change the behavior of a concrete class.
    //Instead of creating concrete objects in UsersController class,
    //we pass the objects to the constructor of UsersController
    public UsersController(IUsersView view, IList users)
    {
        _view = view;
        _users = users;
        view.SetController(this);
    }

    public IList Users
    {
        get { return ArrayList.ReadOnly(_users); }
    }

    private void updateViewDetailValues(User usr)
    {
        _view.FirstName = usr.FirstName;
        _view.LastName  = usr.LastName;
        _view.ID        = usr.ID;
        _view.Department = usr.Department;
        _view.Sex       = usr.Sex;
    }

    private void updateUserWithViewValues(User usr)
    {
        usr.FirstName = _view.FirstName;
        usr.LastName  = _view.LastName;
        usr.ID        = _view.ID;
        usr.Department = _view.Department;
        usr.Sex       = _view.Sex;
    }

    public void LoadView()
    {
        _view.ClearGrid();
        foreach (User usr in _users)
            _view.AddUserToGrid(usr);

        _view.SetSelectedUserInGrid((User)_users[0]);
    }
}

```

```
}

public void SelectedUserChanged(string selectedUserId)
{
    foreach (User usr in this._users)
    {
        if (usr.ID == selectedUserId)
        {
            _selectedUser = usr;
            updateViewDetailValues(usr);
            _view.SetSelectedUserInGrid(usr);
            this._view.CanModifyID = false;
            break;
        }
    }
}

public void AddNewUser()
{
    _selectedUser = new User(" " /*firstname*/,
    " " /*lastname*/,
    " " /*id*/,
    " " /*department*/,
    User.SexOfPerson.Male/*sex*/);

    this.updateViewDetailValues(_selectedUser);
    this._view.CanModifyID = true;
}

public void RemoveUser()
{
    string id = this._view.GetIdOfSelectedUserInGrid();
    User userToRemove = null;

    if (id != "")
    {
        foreach (User usr in this._users)
        {
            if (usr.ID == id)
            {
                userToRemove = usr;
                break;
            }
        }

        if (userToRemove != null)
        {
            int newSelectedIndex = this._users.IndexOf(userToRemove);
            this._users.Remove(userToRemove);
            this._view.RemoveUserFromGrid(userToRemove);

            if (newSelectedIndex > -1 && newSelectedIndex < _users.Count)
            {
                this._view.SetSelectedUserInGrid((User)_users[newSelectedIndex]);
            }
        }
    }
}

public void Save()
{
    updateUserWithViewValues(_selectedUser);
    if (!this._users.Contains(_selectedUser))
    {
        //Add new user
        this._users.Add(_selectedUser);
        this._view.AddUserToGrid(_selectedUser);
    }
    else
}
```

```

{
    //Update existing user
    this._view.UpdateGridWithChangedUser(_selectedUser);
}
_view.SetSelectedUserInGrid(_selectedUser);
this._view.CanModifyID = false;
}
}

```

The controller class is very important and central to the application. It's really important to keep it light, agile and loosely coupled to other components of the program.

Part of View

This section will focus on the scenario of loading the View with the list of users.

As said before our View must implement the `IUsersView` interface. A subset of the implementation is shown in the following code :

```

namespace WinFormMVC.View
{
    public partial class UsersView : Form, IUsersView
    {

```

The `SetController()` member function of `UsersView` allows us to tell the View to which Controller instance it must forward the events and all event handlers simply call the corresponding "event" method on the Controller. As you can see here, `UsersView` also depends on abstractions...

```

public void SetController(UsersController controller)
{
    _controller = controller;
}

```

We also use realisation of several methods from the `IUsersView` interface which use the `User` object:

```

public void AddUserToGrid(User usr)
{
    ListViewItem parent;
    parent = this.grdUsers.Items.Add(usr.ID);
    parent.SubItems.Add(usr.FirstName);
    parent.SubItems.Add(usr.LastName);
    parent.SubItems.Add(usr.Department);
    parent.SubItems.Add(Enum.GetName(typeof(User.SexOfPerson), usr.Sex));
}

public void UpdateGridWithChangedUser(User usr)
{
    ListViewItem rowToUpdate = null;

    foreach (ListViewItem row in this.grdUsers.Items)
    {
        if (row.Text == usr.ID)
        {
            rowToUpdate = row;
        }
    }

    if (rowToUpdate != null)
    {
        rowToUpdate.Text = usr.ID;
        rowToUpdate.SubItems[1].Text = usr.FirstName;
        rowToUpdate.SubItems[2].Text = usr.LastName;
        rowToUpdate.SubItems[3].Text = usr.Department;
        rowToUpdate.SubItems[4].Text = Enum.GetName(typeof(User.SexOfPerson), usr.Sex);
    }
}

```

```
public void RemoveUserFromGrid(User usr)
{
    ListViewItem rowToRemove = null;

    foreach (ListViewItem row in this.grdUsers.Items)
    {
        if (row.Text == usr.ID)
        {
            rowToRemove = row;
        }
    }

    if (rowToRemove != null)
    {
        this.grdUsers.Items.Remove(rowToRemove);
        this.grdUsers.Focus();
    }
}

public string GetIdOfSelectedUserInGrid()
{
    if (this.grdUsers.SelectedItems.Count > 0)
        return this.grdUsers.SelectedItems[0].Text;
    else
        return "";
}

public void SetSelectedUserInGrid(User usr)
{
    foreach (ListViewItem row in this.grdUsers.Items)
    {
        if (row.Text == usr.ID)
        {
            row.Selected = true;
        }
    }
}

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

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

public string ID
{
    get { return this.txtID.Text; }
    set { this.txtID.Text = value; }
}

public string Department
{
    get { return this.txtDepartment.Text; }
    set { this.txtDepartment.Text = value; }
}

public User.SexOfPerson Sex
{
    get
    {
```

```

        if (this.rdMale.Checked)
            return User.SexOfPerson.Male;
        else
            return User.SexOfPerson.Female;
    }
    set
    {
        if (value == User.SexOfPerson.Male)
            this.rdMale.Checked = true;
        else
            this.rdFamele.Checked = true;
    }
}

public bool CanModifyID
{
    set { this.txtID.Enabled = value; }
}

...
}

```

Part of Model

This User class is a Model class. In this example, the User is an extremely simple domain class with no behavior, whereas in a realworld Domain Model you would probably have much more functionality in the domain classes. The model is independent of the user interface. It doesn't know if it's being used from a text-based, graphical or web interface. The Model only holds the in-memory state in a structured format. As you can see, the class contains only private data members and the public interfaces (properties) available to the client code

```

using System;

namespace WinFormMVC.Model
{
    public class User
    {
        public enum SexOfPerson
        {
            Male    = 1,
            Female  = 2
        }

        private string _FirstName;
        public string FirstName
        {
            get { return _FirstName; }
            set
            {
                if (value.Length > 50)
                    Console.WriteLine("Error! FirstName must be less than 51 characters!");
                else
                    _FirstName = value;
            }
        }

        private string _LastName;
        public string LastName
        {
            get { return _LastName; }
            set
            {
                if (value.Length > 50)
                    Console.WriteLine("Error! LastName must be less than 51 characters!");
                else
                    _LastName = value;
            }
        }
    }
}

```



```

private string _ID;
public string ID
{
    get { return _ID; }
    set
    {
        if (value.Length > 9)
            Console.WriteLine("Error! ID must be less than 10 characters!");
        else
            _ID = value;
    }
}

private string _Department;
public string Department
{
    get { return _Department; }
    set { _Department = value; }
}

private SexOfPerson _Sex;
public SexOfPerson Sex
{
    get { return _Sex; }
    set { _Sex = value; }
}

public User(string firstname, string lastname, string id, string department, SexOfPerson
sex)
{
    FirstName = firstname;
    LastName = lastname;
    ID = id;
    Department = department;
    Sex = sex;
}
}
}

```

Part of Client

And now it's good time to show how we use the MVC paradigm effectively, i.e. our code's components of MVC architecture (please see *UseMVCAApplication.csproj*)

```

using System.Collections;
using WinFormMVC.Model;
using WinFormMVC.View;
using WinFormMVC.Controller;

namespace UseMVCAApplication
{
    static class Program
    {
        /// The main entry point for the application.
        [STAThread]
        static void Main()
        {
            //Here we are creating a View
            UsersView view = new UsersView();
            view.Visible = false;

            //Here we are creating a List of users
            IList users = new ArrayList();

```

```

//Here we are add our "commoners" in the list of users
users.Add(new User("Vladimir", "Putin",
"122", "Government of Russia",
User.SexOfPerson.Male));
users.Add(new User("Barack", "Obama",
"123", "Government of USA",
User.SexOfPerson.Male));
users.Add(new User("Stephen", "Harper",
"124", "Government of Canada",
User.SexOfPerson.Male));
users.Add(new User("Jean", "Charest",
"125", "Government of Quebec",
User.SexOfPerson.Male));
users.Add(new User("David", "Cameron",
"126", "Government of United Kingdom",
User.SexOfPerson.Male));
users.Add(new User("Angela", "Merkel",
"127", "Government of Germany",
User.SexOfPerson.Female));
users.Add(new User("Nikolas", "Sarkozy", \
"128", "Government of France",
User.SexOfPerson.Male));
users.Add(new User("Silvio", "Berlusconi",
"129", "Government of Italy",
User.SexOfPerson.Male));
users.Add(new User("Yoshihiko", "Noda",
"130", "Government of Japan",
User.SexOfPerson.Male));

//Here we are creating a Controller and passing two
//parameters: View and list of users (models)
UsersController controller = new UsersController(view, users);
controller.LoadView();
view.ShowDialog();
}
}
}

```

Why is it good?

1. The main advantage of using the MVC pattern is that it makes the code of the user interface more testable
2. It makes a very structured approach onto the process of designing the user interface, which in itself contributes to writing clean, testable code, that will be easy to maintain and extend

The Model-View-Controller is a well-proven design pattern to solve the problem of separating data (model) and user interface (view) concerns, so that changes to the user interface do not affect the data handling, and that the data can be changed without impacting/changing the UI. The MVC solves this problem by decoupling data access and business logic layer from UI and user interaction, by introducing an intermediate component: the controller. This MVC architecture enables the creation of reusable components within a flexible program design (components can be easily modified)

History

- 10th Mayl 2012: Initial post

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author



Volynsky Alex

Software Developer

Canada 

Mr.Volynsky Alex is a Software Engineer in a leading software company. Alex is skilled in many areas of computer science. He has over 14 years of experience in the design & development of applications using C/C++/STL, Python, Qt, MFC, DirectShow, JavaScript, VBScript, Bash and of course - C#/.NET.

In addition, Alex is the [active member of Intel® Developer Zone](#) (he was awarded by Intel® Green Belt for his active contribution to the Intel Developer Zone community for developers using Intel technology).

Alex is also interested in the Objective-C development for the iPad/iPhone platforms and he is the developer of the free [15-puzzle game](#) on the App Store.

Overall, Alex is very easy to work with. He adapts to new systems and technology while performing complete problem definition research.

His hobbies include yacht racing, photography and reading in multiple genres.

He is also fascinated by attending computer meetings in general, loves traveling, and also takes pleasure in exercising and relaxing with friends.

Visit his C++ 11 [blog](#)

Comments and Discussions

 **155 messages** have been posted for this article Visit <https://www.codeproject.com/Articles/383153/The-Model-View-Controller-MVC-Pattern-with-Csharp> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#)

[Advertise](#)

[Privacy](#)

[Cookies](#)

[Terms of Use](#)

Article Copyright 2012 by Volynsky Alex
Everything else Copyright © [CodeProject](#), 1999-2020

Web01 2.8.200602.2