An Architecture and Implement Model for Model-View-Presenter Pattern

Yang Zhang
School of Computer Science and Engineering
Beihang University
Beijing, China
ericzhang.buaa@gmail.com

Yanjing Luo
School of Computer Science and Engineering
Beihang University
Beijing, China
luo_yanjing@sina.com

***Abstract*—Model-View-Presenter (MVP) is a pattern which aimed at providing a cleaner separation between the View, the Model and the Presenter. The paper advances an architecture model of MVP pattern on .NET platform and a formal method of how to implement it. After that, an example of implement MVP pattern to web application or desktop application is cited.**

Keywords- mvp; pettern; architecture; presentation layer;

I. INTRODUCTION

Model-View-Presenter (MVP) is an architecture pattern for the presentation layer of software applications. The pattern was originally developed at Taligent in 1990s [1] and first was implemented in C++ and Java.

In MVP, the View and the Model are neatly separated and the View exposes a contract through which the Presenter access the portion of View that is dependent on the rest of the system.

The Model is the component which preserves data, state and business logic; it just exposes a group of service interfaces to Presenter and hides the internal details.

The View is the user interface, it receives user's action and contract to Presenter to achieve user's need, and then the View responds user by result information.

The Presenter sits in between the View and the Model; it receives input from the View and passes commands down to the Model. It then gets result and updates the View trough the contracted View interface.

"Fig. 1" illustrates the parts of MVP pattern and how they interact with each other.

Since the MVP pattern was put up in 1990s, it has been widely discussed in the area of software engineering; Martin Fowler reported some methods of implementing MVP at his papers [5] and books [6]. However, few wittier have considered how to implement it on concrete program; this process is extremely dependent on experience of developers.

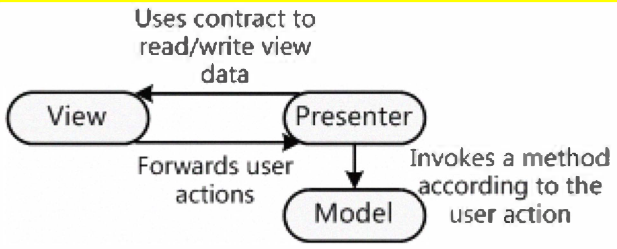

Uses contract to read/write view data
View
Presenter
Forwards user actions
Invokes a method according to the user action
Model

Figure 1. MVP pattern.

Contrast to traditional presentation layer, the advantage of presentation layer with MVP pattern is based on tree facts:

- The View doesn't know the Model. Because of this, there is a low coupling between Model and View. It means that if Model or View was changed, another part not needs to modify as long as interfaces are stable. This also stands for the flexibility of architecture and the reusability of business logic in Model.
- The Presenter ignores any UI technology behind the View. According to this, the replacement of UI technology, such as transfer Windows Forms to WPF or to Web Forms, is not need any change of other parts. Even one application could have more than one UI technologies but one Model so that the C/S deployment and the B/S deployment are supported by it at the same time.
- The View is mockable for testing purposes. In tradition, it is impossible to test View or business logic component before another has completed because of the tight coupling between View and business logic. By the same token, the unit testing for View or business logic component is difficult. All of those problems are solved by MVP pattern. In MVP, there is no direct dependency between View and Model. For that reason, developer could use mock object to inject into View or Model so that they can be tested on one's own.

II. THE ARCHITECTURE MODEL

As a pattern, MVP has various expressive forms and architectures when implementing in different platforms, the concrete implement is restrained by the features of platform, and consistency is another factor which should be considered when designing the concrete architecture model.

Based on the above, an architecture model of implementing MVP pattern on .NET which is illustrated by "Fig. 2" is given here. This architecture model not only takes advantage of many specific characters of OOAD, practical experience also proves that this is workable and well-behaved.

This model is made up by five parts:

IView is the abstraction of View that is composed of a set of rules that declare what data and functions should be implemented in View. Generally every View component has its own IView component.

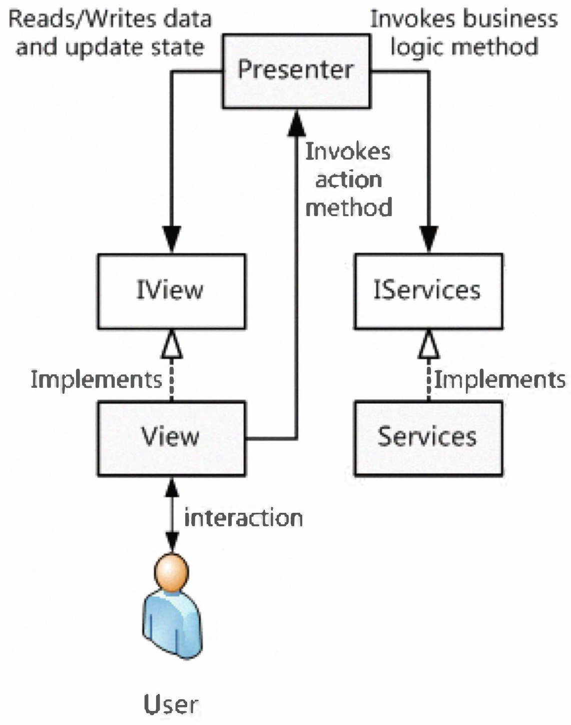


Figure 2. The architecture model of MVP on .NET.

View is the part which interacts with users. There are many technologies could be used to implement the View on .NET platform, such as Windows Forms, Web Forms, Silverlight, WPF and so on. Every View component should implement the homologous IView and one IView component could has many implements with different UI technologies, as a result the Views implemented from the same IView could take the place of each other.

IServices is a set of interfaces that define the functions should be implemented by business logic components. It is correspond to interface of The Model.

Services are the business logic components that implements IServices. View and Presenter need the help of Services to do application business on account of they do not have any business logic. Services having nothing to do UI technology commonly, the concrete implement of them are some general classes. Sometimes Services need a Repository to deal with the operation of database, this is out of this paper's range.

Presenter is the core of MVP pattern. On .NET platform presenter is a number of classes that dependent on IView and IServices. Just like the Services, Presenter is foreign to UI technology because it just dependent to IView. Presenter receives user actions and read input data from view, and then it invokes functions in Services to complete the business logic and modifies View's state. This entire works are called presentation logic.

"Fig. 3" is the sequence diagram that shows how MVP pattern works.

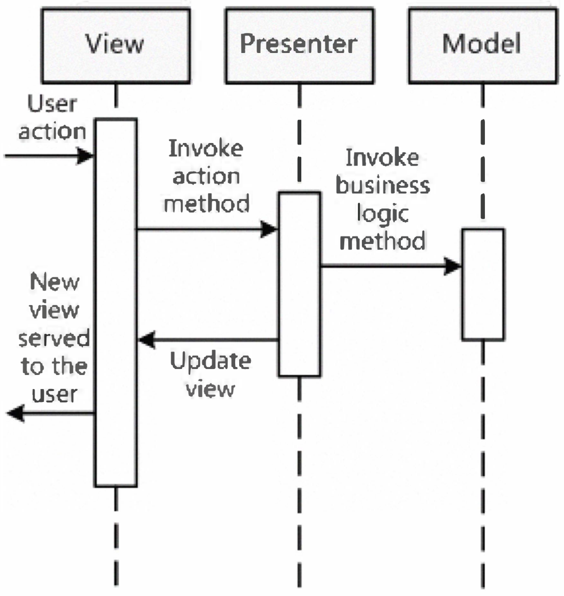


Figure 3. Sequence diagram of MVP pattern.

III. Method of Implementing MVP

Generally, the method of how to implement a pattern is described by natural language, but a kind of formalistic description based on set theory will be used to explain the procedures of implementing MVP pattern on .NET platform in this paper.

Above all, there are some sets, functions and formulas need to be defined:

***Let** I = {i_1, i_2, ..., i_n}. It is the set of input data that each element is input by user.*

***Let** O = {o_1, o_2, ..., o_m}. It is the set of output data that each element should be output to View.*

***Let** A = {a_1, a_2, ..., a_k}. It is the set of actions that each element indicates an action that user could act on View.*

***Let** VAM = <I, O, A>. It is View Abstract Model (VAM) that indicates the abstraction of a view component. Unlike I, O and A, VAM is not a set but an ordered triple.*

Then, the implement of MVP could be decomposed into five steps:

1) Building UI prototype.

The task of this step is to build prototype for every UI component.

Prototype is the visual element or text description that indicates the frame structure of UI component. A UI component is a part of UI which is fully-formed and independent; it often exists as a windows form or a web page. Every UI component has its own prototype.

There are various methods to build UI prototype such as drawing UI blueprint, developing UI prototype program and describing UI by text.

2) Creating the View Abstract Model.

According to the definition of VAM, the task of creating it is divided into creating set *I*, creating set *O* and creating set *A*, that is, to find every input data, every output data and every action user may act on View.

Just like UI prototype, every UI component has its own VAM, so this step's assignment is to find all of the input data, output data and actions at UI prototype which was built in step 1.

3) Defining interfaces of View and Presenter.

Formally, interfaces of View and Presenter are sets that include function definition elements.

A function definition element is an ordered triple just like *<N, R, P>*. *N* is the name of function, *R* is the return type of function and *P* is a set with parameter elements. A function definition element is an abstraction of a function.

The algorithm of define interfaces of View and Presenter as follows:

```
CREATE-IVIEW-AND-IPRESENTER (I, O, A)
  1.  Let getter(data) is to define a getter function for data.
  2.  Let setter(data) is to define a setter function for data.
  3.  Let handle(action)is to define a function that handle the action.
  4.  Let IView = {} is the set indicates interface of View.
  5.  Let IPresenter = {} is the set indicates interface of Presenter.
  6.  For each element in I
          Add getter(element) to IView.
      End for
  7.  For each element in O
          Add setter(element) to IView.
      End for
  8.  For each element in A
          Add handle(element) to IPresenter.
      End for
```

It must be explained that also IPresenter is created here, it needs not to implement in program on account of it just intermediate product that helps to create Presenter. But, IView should be implemented in program.

4) Implementing View and Presenter.

IView and IPresenter set the rules for View and Presenter. The job of this step is to implement the View and Presenter on the basis of IView and IPresenter.

5) Defining services interfaces and implement services.

In the process of implementing Presenter, it will be find that many action handle functions need business logic functions, this demand could be abstracted as services interfaces and business logic components that should be implemented based on this interfaces.

IV. Example

In order to explain how to practice the model and method above, an example is given in this chapter.

Think about a B2C e- commerce system, customers could browse, search and buy products online with it. One of its feature is customer could add product to cart: system showing name and price of target product to customer when a product is selected and once customer has changed product's amount, system updates the total money that customer should to pay.

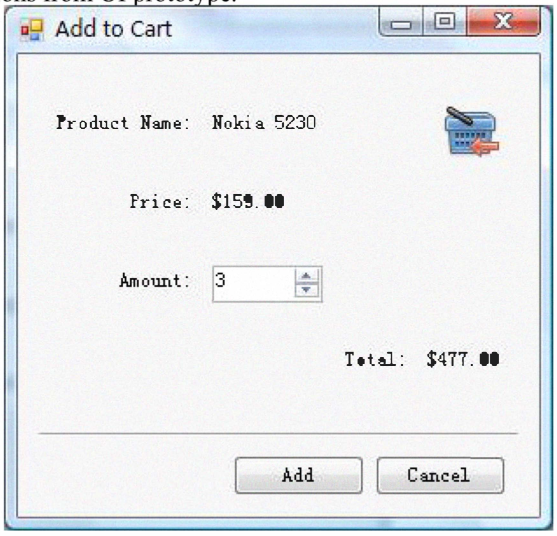
This feature will be the example that shows how to implement MVP pattern on .NET platform with the architectural model and method above.

1) Building UI prototype.

Firstly, a UI prototype which is illustrated by "Fig. 4" was developed on basis of description of this feature. It has not any logic but just a frame of UI.

2) Creating the View Abstract Model.

Next task is to identify all input data, output data and actions from UI prototype.



Product Name: Nokia 5230

Price: $159.00

Amount: 3

Total: $477.00

Figure 4. The UI prototype.

Input data is data that input by user and transferred from View to Presenter, output data is data that transferred from Presenter to View and displayed to user, actions is the behaviors that users could act on UI prototype.

"Fig. 5" demonstrates all data and actions found in UI prototype. As shown in the figure:

- *Amount* is the only input data in this UI.
- *Product Name*, *Price* and *Total* are output data that showed to user.
- User's actions on this UI include *Add Product to Cart* (when click "Add" button), *Cancel* (when click "Cancel" button), *Change Amount* (when click arrow buttons or change number text).

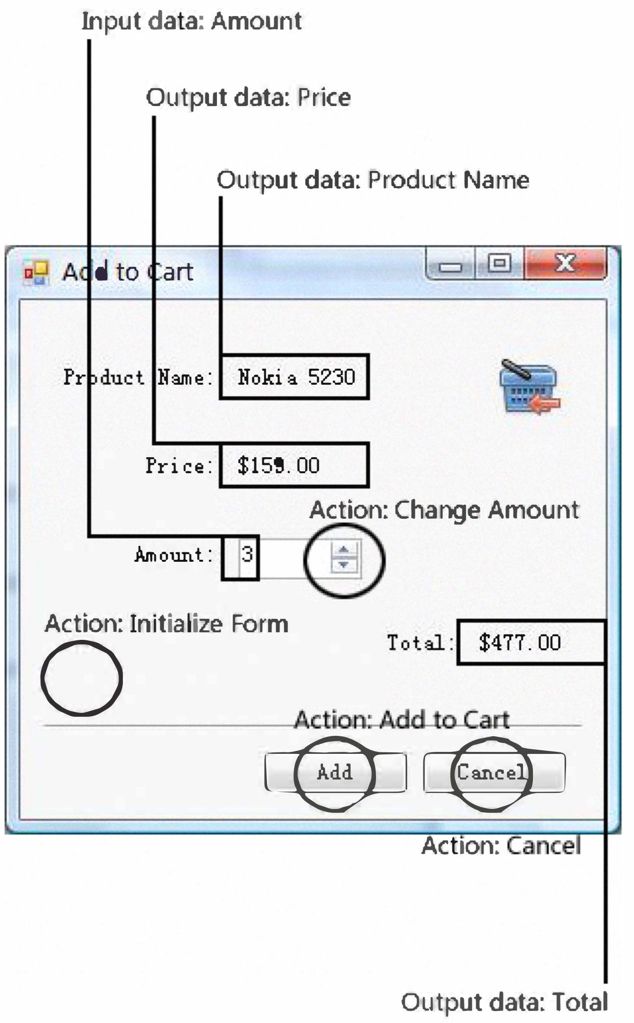


Figure 5. Find input/output data and actions.

Worth mentioning, there is an inconspicuous action, *Initialize Form*, that occurs when user opening this UI component.

From the above, VAM of this UI component is following:

I = {Amount}

O = {Product Name, Price, Total}

A = {Add Product to Cart, Cancel, Change Amount, Initialize Form}

VAM

= <I, O, A>

= < {Amount}, {Product Name, Price, Total}, {Add Product to Cart, Cancel, Change Amount, Initialize Form} >

3) Defining interfaces of View and Presenter.

IView and IPresenter could obtain by applying *CREATE-IVIEW-AND-IPRESENTER* to VAM built above. Details are listed in "Table I" and "Table II".

Generally, getter function has no parameter and its return type is data's. Setter function has one parameter with the same type of data and a void return value. Action functions have no parameter but a void return value.

4) Implementing View and Presenter.

After defined interfaces, it is not difficult to implement them.

For View, the implement is developing UI component with a kind of UI technology and implementing all functions that "Table I" lists.

Presenter will be implemented as a class that includes all action functions.

Getter and setter functions are so simple that it has no use for explaining in detail. So this paper will only describe the action functions.

TABLE I. VIEW INTERFACE FUNCTIONS

Name	*Return Type*	*Parameters*
getAmount	decimal	{}
setProductName	void	{name}
setPrice	void	{price}
setTotal	void	{total}

TABLE II. PRESENTER INTERFACE FUNCTIONS

Name	*Return Type*	*Parameters*
AddAction	void	{}
CancelAction	void	{}
ChangeAmountAciton	void	{}
InitializeAction	void	{}

The logics of action functions are listed hereinafter.

- The task of *AddAction* is to read the *Amount* user input and save it in database with product and customer's information.
- *CancelAction* is very simple that only closes the window.

- *ChangeAmountAction* is triggered when user changed the *Amount* of product. In this function, the code should multiply *Price* by new *Amount* read from UI and outputs the result to *Total* immediately.
- *InitializeAction*'s logic is to output *Product Name*, *Price*, default *Amount* and default *Total* to UI.

Additionally, how View and Presenter interact is an important problem to solve.

Although a great deal of approaches could solve this problem, one of them is recommended strongly that declare a private member with the type of IView in Presenter and not to instantiate it but to expose a constructor injection point. This ensures the Presenter independent of any concrete UI component; in other words, every UI component who implemented the IView could use this Presenter. UI component also has a private member with the type of Presenter, unlike Presenter, UI component instantiates this member at constructor and uses itself as the parameter to inject into Presenter object. What follows in the passage are code structures of this approach.

Presenter Code Structure

```
public class Presenter
{
        private IView _view;

        public Presenter (IView view)
        {
                this._view = view;
        }

        /* Action Functions */
}
```

UI Code Structure

```
public class View : IView
{
        private Presenter _presenter;

        /* IView Members */

        public View()
        {
                This._presenter = new Presenter (this);
        }

        /* UI Technology Codes */
}
```

The code structures are written in C# that is a popular language on .NET platform. Nevertheless, they are true of Java, C++ and other object-oriented languages too.

5) Defining services interfaces and implement services.

Because this step is not the job of presentation layer, the details is omitted here.

V. SUMMARY

This paper describes an architectural model and a formal method of implementing MVP pattern on .NET platform. For the purpose of making them easy to understand and use, the example of an e- commerce system was cited that demonstrates how to put the architecture model and formal method into practice. Although the model and method in this paper is implemented on .NET, it suit to other platforms and languages as well. Additionally, the model and method is fit to both desktop application and web application, in other words, they are UI technology independence.

REFERENCES

[1] Mike Potel, “MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java”, Taligent Inc. 1996.

[2] Steve Burbeck, “Applications Programming in Smalltalk-80: How to use Model-View-Controller (MVC)”, st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html.

[3] Andy Bower, Blair McGlashan, "Twisting the Triad: The evolution of the Dolphin Smalltalk MVP application framework", European Smalltalk User Group (ESUG), 2000.

[4] Dino Esposito, Andrea Saltarello, “Microsoft .NET: Architecting Applications for the Enterprise” , Microsoft Press, 2008.

[5] Martin Fowler, "Model View Presenter", www.martinfowler.com/eaaDev/ModelViewPresenter.html, July 2004.

[6] Martin Fowler, “Patterns of Enterprise Application Architecture”, Addison-Wesley Professional, November 2002.

[7] Jeffrey Richter, “CLR via C#”, Microsoft Press, 2010.

[8] Micah Alles, David Crosby, Brian Harleton, Greg Pattison, Carl Erickson, Michael Marsiglia, Curt Stienstra, “Presenter First: Organizing Complex GUI Applications for Test-Driven Development”, Agile Conference, 2006.

[9] Roger Took, “Surface Internation: A Paradigm and Model for Separating Application and Interface”, 1990 ACM O-89791-345-O/90/0004-0035, pp. 35-42.

[10] Trygve Reenskaug, "MODELS-VIEWS-CONTROLLERS", Technical note, Xerox PARC, December 1979.

[11] Michael Feathers, "The Humble Dialog Box", Object Mentor, 2002.

[12] Martin Fowler, “Mocks Aren’t Stubs”, www.martinfowler.com/articles/mocksArentStubs.html, July 2004.

[13] Spring.NET Application Framework, www.springframework.net/