


# MVC or MVP Pattern – Whats the difference?

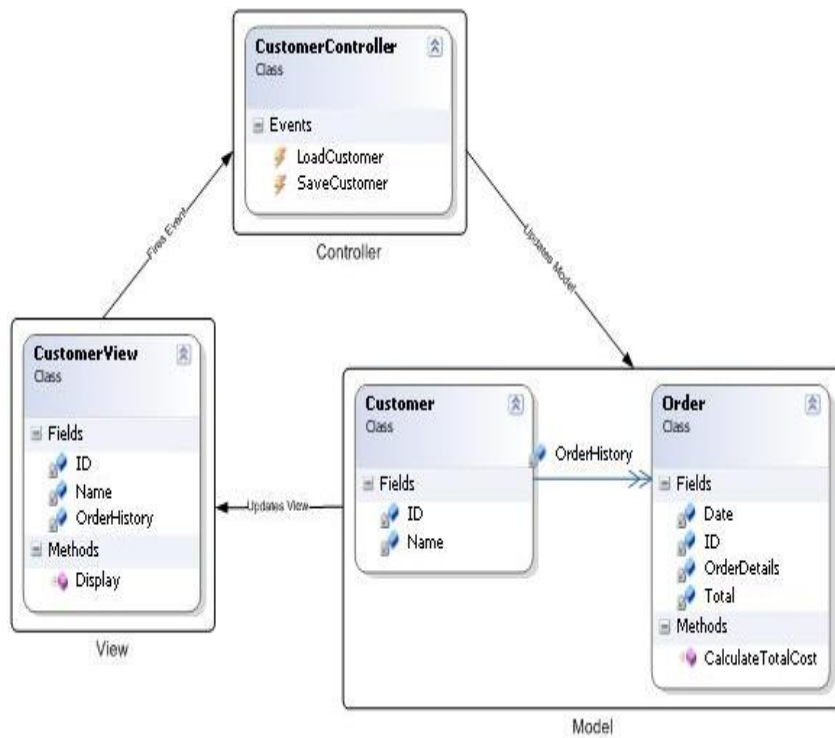
 [Log in](#) to  
like this  
post!

[Todd Snyder](#) / Wednesday, October 17, 2007

Over the years I have mentored many developers on using design patterns and best practices. One question that keeps coming up over and over again is: What are the differences between the Model View Controller (MVC) and Model View Presenter (MVP) patterns? Surprisingly the answer is more complex than what you would suspect. Part of reasons I think many developers shy away from using either pattern is the confusion over the differences.

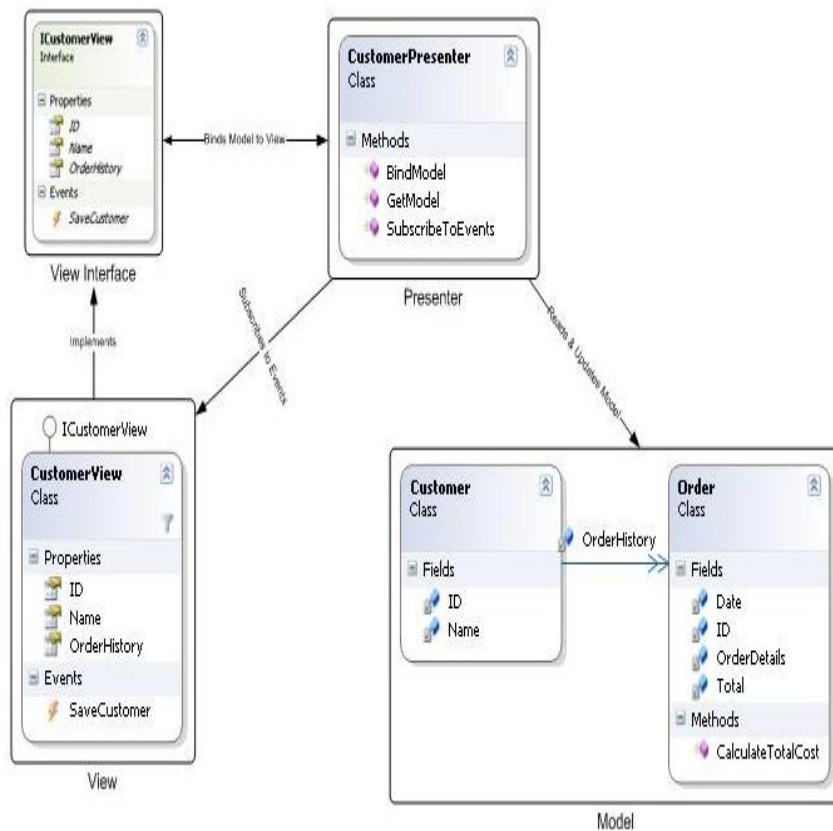
Before we dig into the differences let's examine how the patterns work and the key benefits to using either one. Both (MVC & MVP) patterns have been used for several years and address a key OO principal namely separation of concerns between the UI and the business layers. There are a number of frameworks is used today that based on these patterns including [JAVA Struts](#), [ROR](#), [Microsoft Smart Client Software Factory \(CAB\)](#), [Microsoft Web Client Software Factory](#), and the recently announced [ASP.Net MVC framework](#).

## Model View Controller (MVC) Pattern



The MVC pattern is a UI presentation pattern that focuses on separating the UI (View) from its business layer (Model). The pattern separates responsibilities across three components: the view is responsible for rendering UI elements, the controller is responsible for responding to UI actions, and the model is responsible for business behaviors and state management. In most implementations, all three components can directly interact with each other and in some implementations, the controller is responsible for determining which view to display ([Front Controller Pattern](#)),

### Model View Presenter (MVP) Pattern



The MVP pattern is a UI presentation pattern based on the concepts of the MVC pattern. The pattern separates responsibilities across four components: the view is responsible for rendering UI elements, the view interface is used to loosely couple the presenter from its view, the presenter is responsible for interacting between the view/model, and the model is responsible for business behaviors and state management. In some implementations, the presenter interacts with a service (controller) layer to retrieve/persist the model. The view interface and service layer are commonly used to make writing unit tests for the presenter and the model easier.

### Key Benefits

Before using any pattern a developer needs to consider the pros and cons of using it. There are a number of key benefits to using either the MVC or MVP pattern (See the list below). But, there also a few drawbacks to consider. The biggest drawbacks are adding complexity and learning curve. While the patterns may not be appropriate for simple solutions; advance solutions can greatly benefit from using the pattern. I'm my experience a have seen a few solutions eliminate a large amount of complexity but being refactored to use either pattern.

- Loose coupling – The presenter/controller are an intermediary between the UI code and the model. This allows the view and the model to evolve independently of each other.
- Clear separation of concerns/responsibility
  - o UI (Form or Page) – Responsible for rendering UI elements
  - o Presenter/controller – Responsible for reacting to UI events and interacts with the model
  - o Model – Responsible for business behaviors and state management
- Test Driven – By isolating each major component (UI, Presenter/controller, and model) it is easier to write unit tests. This is especially true when using the MVP pattern which only interacts with the view using an interface.
- Code Reuse – By using a separation of concerns/responsible design approach you will increase code reuse. This is especially true when using a full-blown domain model and keeping all the business/state management logic where it belongs.
- Hide Data Access – Using these patterns forces you to put the data access code where it belongs in a data access layer. There a number of other patterns that typically works with the MVP/MVC pattern for data access. Two of the most common ones are repository and unit of work. (See Martin Fowler – Patterns of Enterprise Application Architecture for more details)
- Flexibility/Adaptable – By isolating most of your code into the presenter/controller and model components your code base is more adaptable to change. For example, consider how much UI and data access technologies have changed over the years and the number of choices we have available today. A properly design solution using MVC or MVP can support multi UI and data access technologies at the same time.

### Key Differences

So what really are the differences between the MVC and MVP pattern. Actually, there are not a whole lot of differences between them. Both patterns focus on separating responsibility across multi components and promote loosely coupling the UI (View) from the business layer (Model). The major differences are how the pattern is implemented and in some advanced scenarios, you need both presenters and controllers.

Here are the key differences between the patterns:

- MVP Pattern
  - o The view is more loosely coupled to the model. The presenter is responsible for binding the model to the view.
  - o Easier to unit test because interaction with the view is through an interface
  - o Usually, view presenter map one to one. Complex views may have multi presenters.
- MVC Pattern
  - o Controllers are based on behaviors and can be shared across views
  - o Can be responsible for determining which view to display ([Front Controller Pattern](#))

Hopefully, you found this post interesting and it helped clarify the differences between the MVC and MVP pattern. If not, do not be discouraged patterns are powerful tools that can be hard to use sometimes. One thing to remember is that a pattern is a blueprint and not an out of the box solutions. Developers should use them as a guide and modify the implementation according to their problem domain.



[Log in](#) to like this post!



 [OOD](#), [Best Practices](#), [Design Patterns](#)

CHAT