MVC and MVP

References

- http://www.martinfowler.com/eaaDev/ModelViewPresenter.html
- http://www.codeproject.com/Articles/42830/Model-View-Controller-Model-View-Presenter-and-Mod
- http://www.infragistics.com/community/blogs/todd_snyder/archive/2007/10/17/mvc-or-mvp-pattern-whats-the-difference.aspx
- http://www.javacodegeeks.com/2012/02/gwt-mvp-made-simple.html
- http://msdn.microsoft.com/en-us/library/ff647543.aspx

Overview

- MVC - Model-View-Controller
  - One of the most quoted patterns
  - designed by Trygve Reenskaug, a Norwegian computer engineer, while working on Smalltalk-80 in 1979 .
  - was subsequently described in depth in the highly influential “Design Patterns: Elements of Reusable Object-Oriented Software” [2], a.k.a. the “Gang of Four” book, in 1994.
- MVP
  - Was aimed to address the shortfalls of the MVC.
  - Two years later, Mike Potel from Taligent (IBM) published his paper, “Model-View-Presenter (MVP) - The Taligent Programming Model for C++ and Java” [3],
- Both MVP and MVC
  - have since been widely adopted by Microsoft in their Composite Application Blocks (CAB) and the ASP.NET MVC frameworks.

MVC

- is not referred to as a design pattern
  - "Gang of Four"
  - but a "set of classes to build a user interface"
  - uses design patterns such as Observer, Strategy, and Composite.
  - also uses Factory Method and Decorator,
    - the main MVC relationship is defined by the Observer and Strategy patterns.
- There are three types of objects.
  - The Model
    - is our application data,
  - the View
    - is a screen, and
  - the Controller defines
    - the way the View reacts to user input.
  - The views and models use the
    - Publish-Subscribe protocol - when Model data is changed, it will update the View.
    - It allows us to attach multiple Views to the same Model. This is achieved by using the Observer design pattern.

Observer design pattern

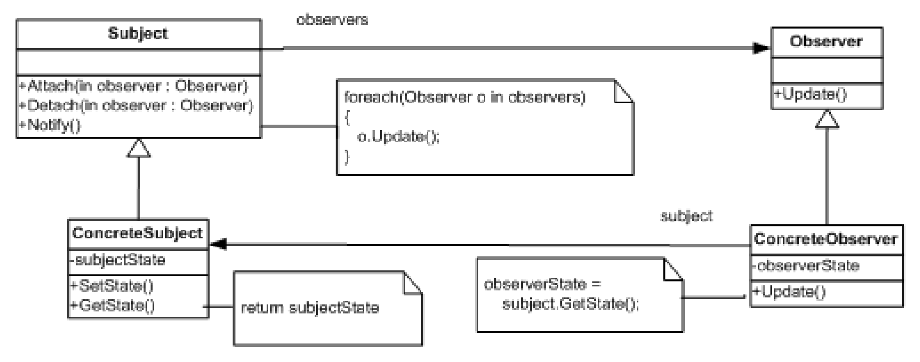


Figure 1: Observer

```
public abstract class Subject
{
    private readonly ICollection<Observer> Observers =
            new Collection<Observer>();

    public void Attach(Observer observer)
    {
        Observers.Add(observer);
    }
    public void Detach(Observer observer)
    {
        Observers.Remove(observer);
    }
    public void Notify()
    {
        foreach (Observer o in Observers)
        {
            o.Update();
        }
    }
}

public class ConcreteSubject : Subject
{
    public object SubjectState { get; set; }
}

public abstract class Observer
{
    public abstract void Update();
}

public class ConcreteObserver : Observer
{
    private object ObserverState;
    private ConcreteSubject Subject { get; set; }

    public ConcreteObserver(ConcreteSubject subject)
    {
        Subject = subject;
    }
    public override void Update()
    {
        ObserverState = Subject.SubjectState;
    }
}
```

Strategy design pattern.

View-Controller relationship of MVC.

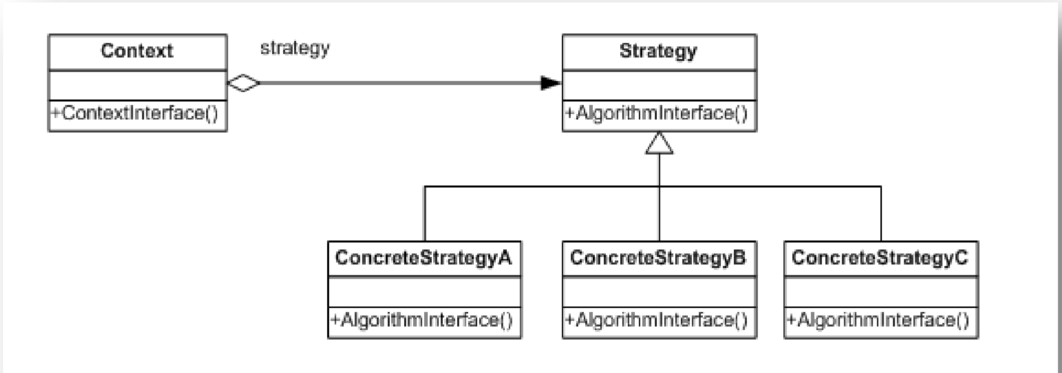
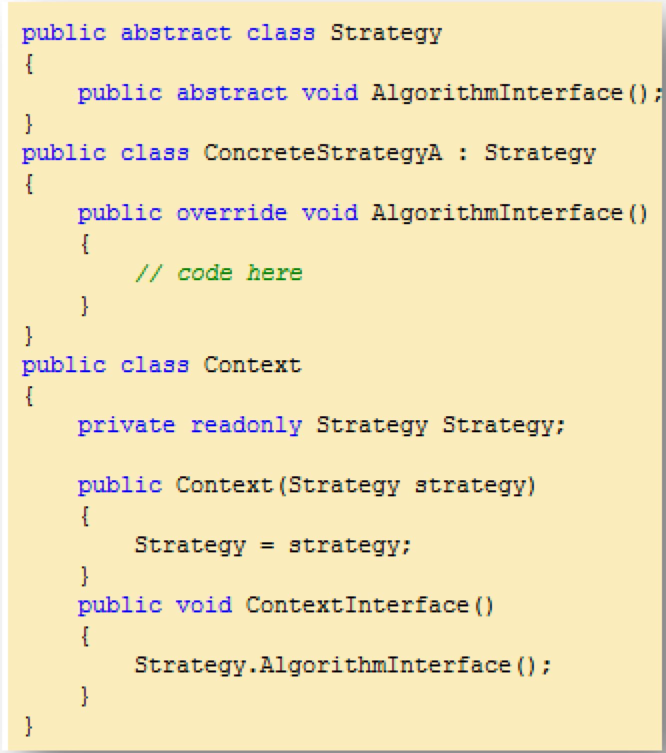


Figure 2: Strategy



```
public abstract class Strategy
{
    public abstract void AlgorithmInterface();
}
public class ConcreteStrategyA : Strategy
{
    public override void AlgorithmInterface()
    {
        // code here
    }
}
public class Context
{
    private readonly Strategy Strategy;

    public Context(Strategy strategy)
    {
        Strategy = strategy;
    }
    public void ContextInterface()
    {
        Strategy.AlgorithmInterface();
    }
}
```

MVC

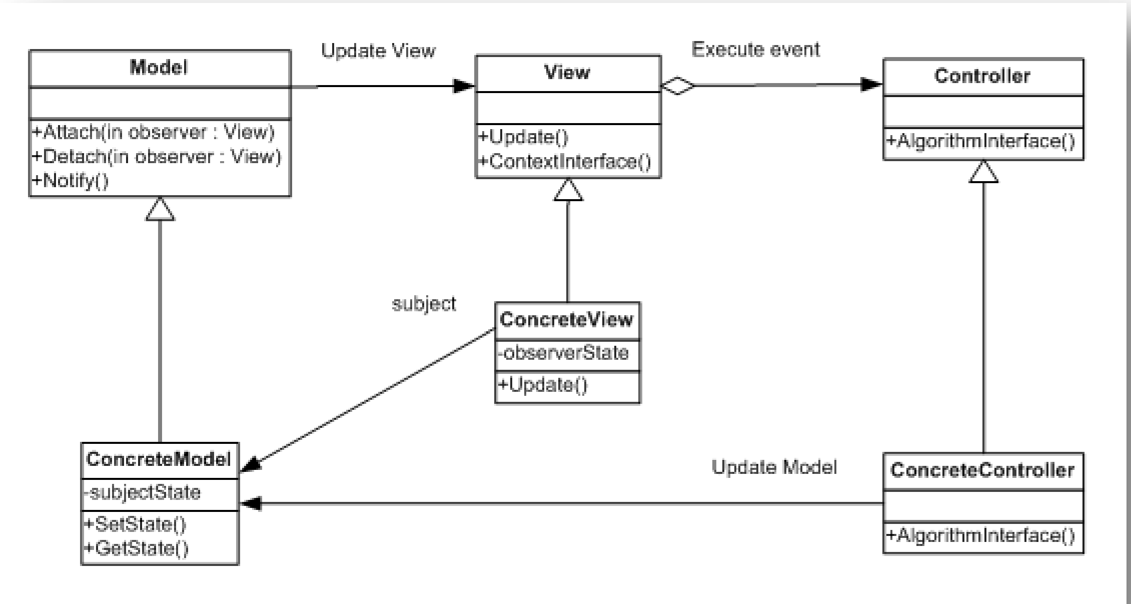
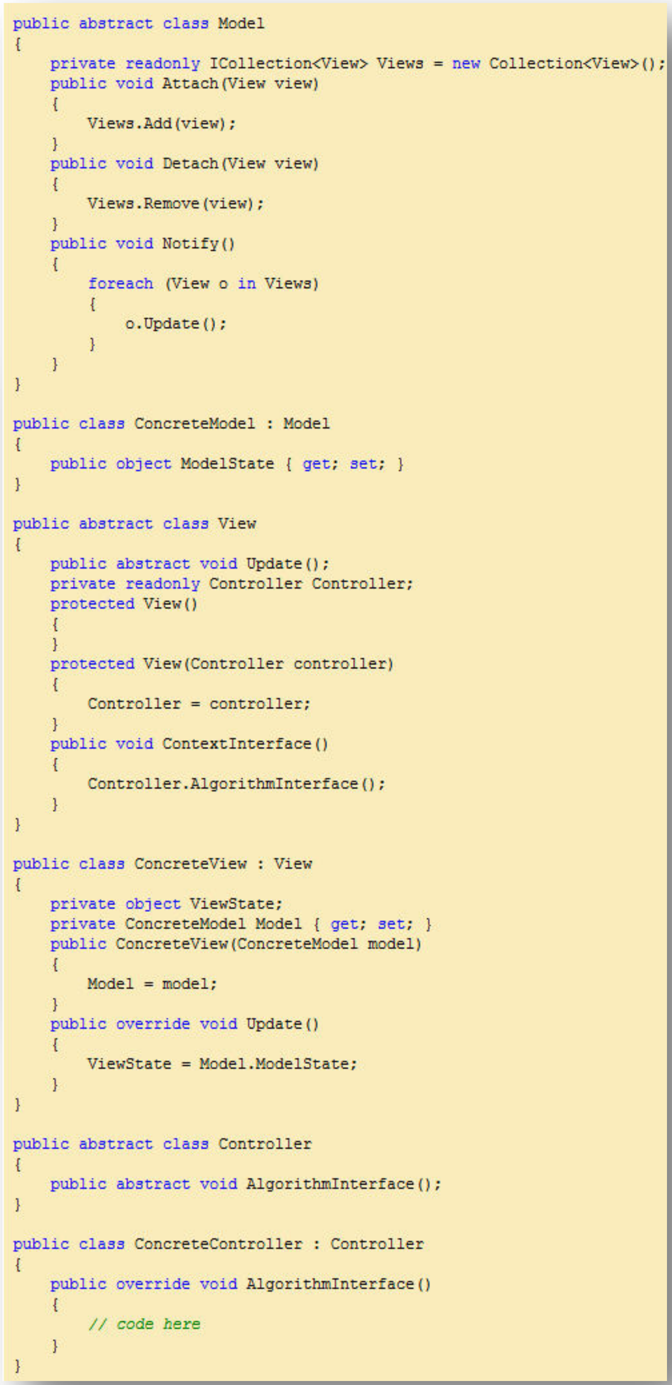


Figure 3: MVC



```
public abstract class Model
{
    private readonly ICollection<View> Views = new Collection<View>();
    public void Attach(View view)
    {
        Views.Add(view);
    }
    public void Detach(View view)
    {
        Views.Remove(view);
    }
    public void Notify()
    {
        foreach (View o in Views)
        {
            o.Update();
        }
    }
}

public class ConcreteModel : Model
{
    public object ModelState { get; set; }
}

public abstract class View
{
    public abstract void Update();
    private readonly Controller Controller;
    protected View()
    {
    }
    protected View(Controller controller)
    {
        Controller = controller;
    }
    public void ContextInterface()
    {
        Controller.AlgorithmInterface();
    }
}

public class ConcreteView : View
{
    private object ViewState;
    private ConcreteModel Model { get; set; }
    public ConcreteView(ConcreteModel model)
    {
        Model = model;
    }
    public override void Update()
    {
        ViewState = Model.ModelState;
    }
}

public abstract class Controller
{
    public abstract void AlgorithmInterface();
}

public class ConcreteController : Controller
{
    public override void AlgorithmInterface()
    {
        // code here
    }
}
```

MVC Diagram

- If we leave out the concrete classes for simplicity, we will get a more familiar MVC diagram.
- Please note that we use pseudo- rather than proper UML shapes, where circles represent a group of classes (e.g., Model and ConcreteModel), not classes as on the UML class diagram on Figure 3.

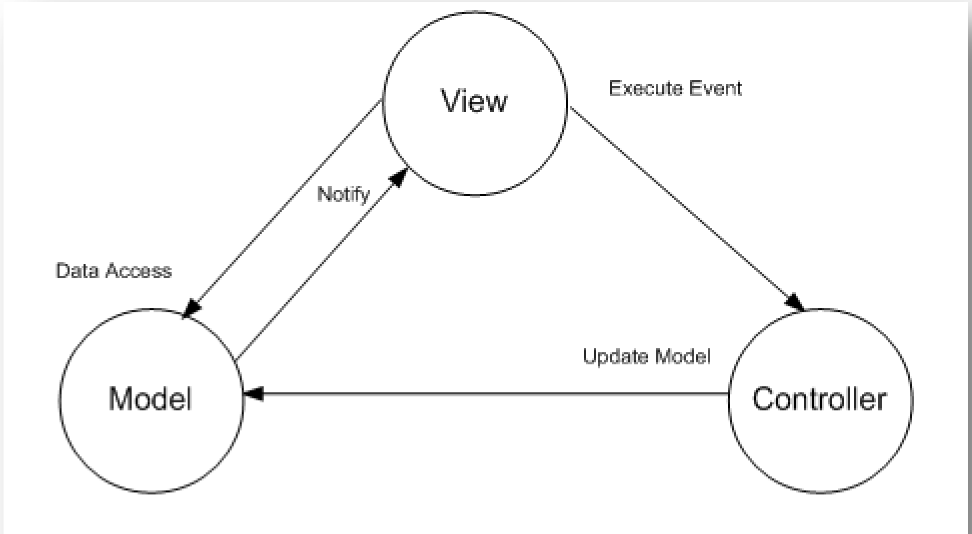
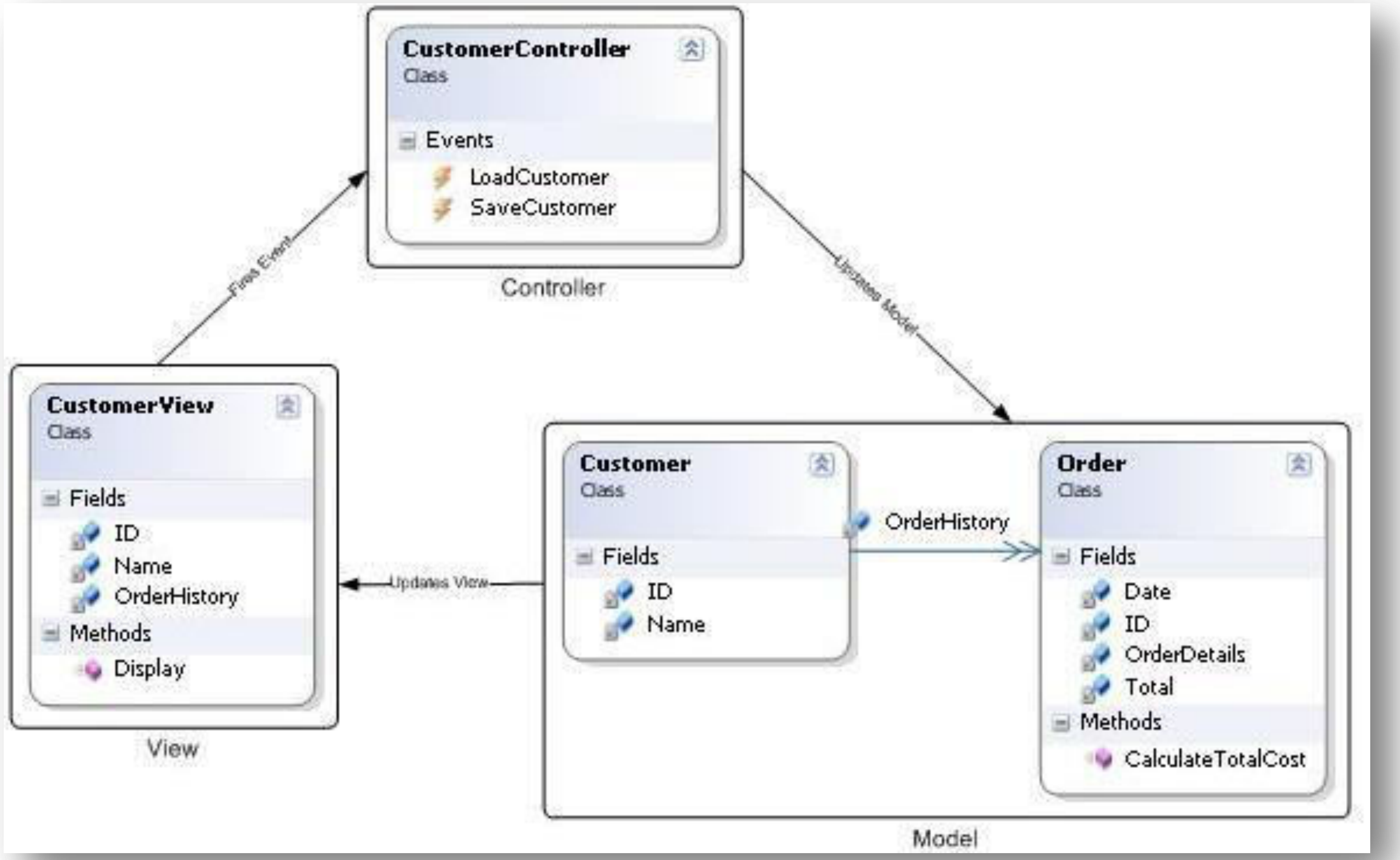


Figure 4: Simplistic MVC

Example


CustomerController
Class
Events
LoadCustomer
SaveCustomer
Controller
Fires Event
Updates Model
CustomerView
Class
Fields
ID
Name
OrderHistory
Methods
Display
View
Updates View
Customer
Class
Fields
ID
Name
OrderHistory
Order
Class
Fields
Date
ID
OrderDetails
Total
Methods
CalculateTotalCost
Model

MVP

- **Scenario**
  - A page in a Web application contains controls that display application domain data.
  - A user can
    - modify the data and
    - submit the changes.
  - The page
    - retrieves the domain data,
    - handles user events,
    - alters other controls on the page in response to the events, and
    - submits the changed domain data.
- **Problem**
  - Including the code that performs these functions in the Web page makes the class complex, difficult to maintain, and hard to test.
  - In addition, it is difficult to share code between Web pages that require the same behavior.

Forces

- You want to maximize the code that can be tested with automation. (Views are hard to test.)
- You want to share code between pages that require the same behavior.
- You want to separate business logic from UI logic to make the code easier to understand and maintain.

Solution

- Separate the responsibilities
  - view : for the visual display and
  - Presenter: the event handling
- Collaboration
  - The view class (the Web page) manages the controls on the page and forwards user events to a presenter class.
  - The presenter contains the logic to respond to the events, update the model (business logic and data of the application) and, in turn, manipulate the state of the view.
- Presenter have a reference to the view interface
  - Not concrete implementation of the view
  - to facilitate testing the presenter,.
  - By doing this, you can easily replace the real view with a mock implementation to run tests.

MVP

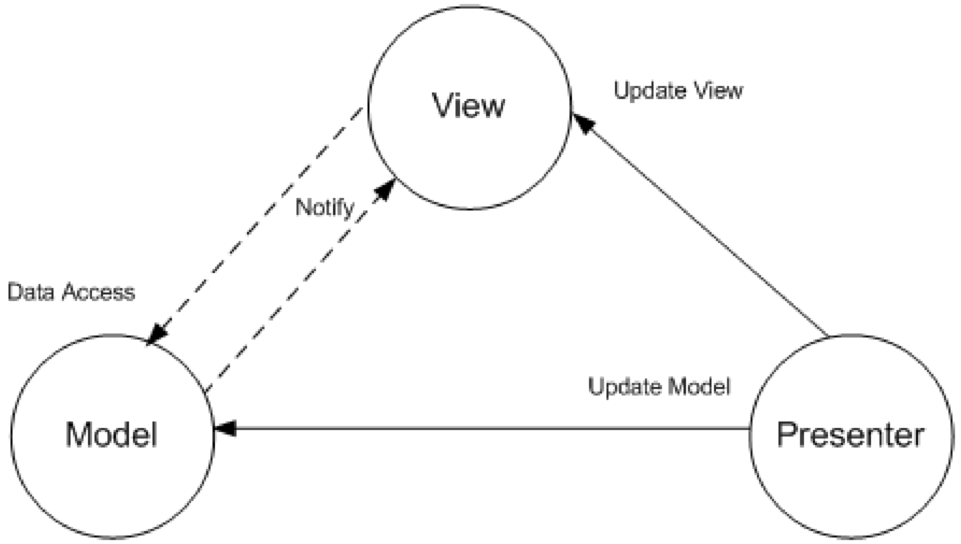


Figure 5: Simplistic MVP

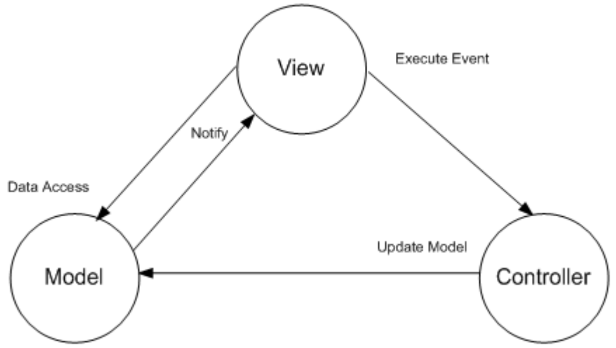


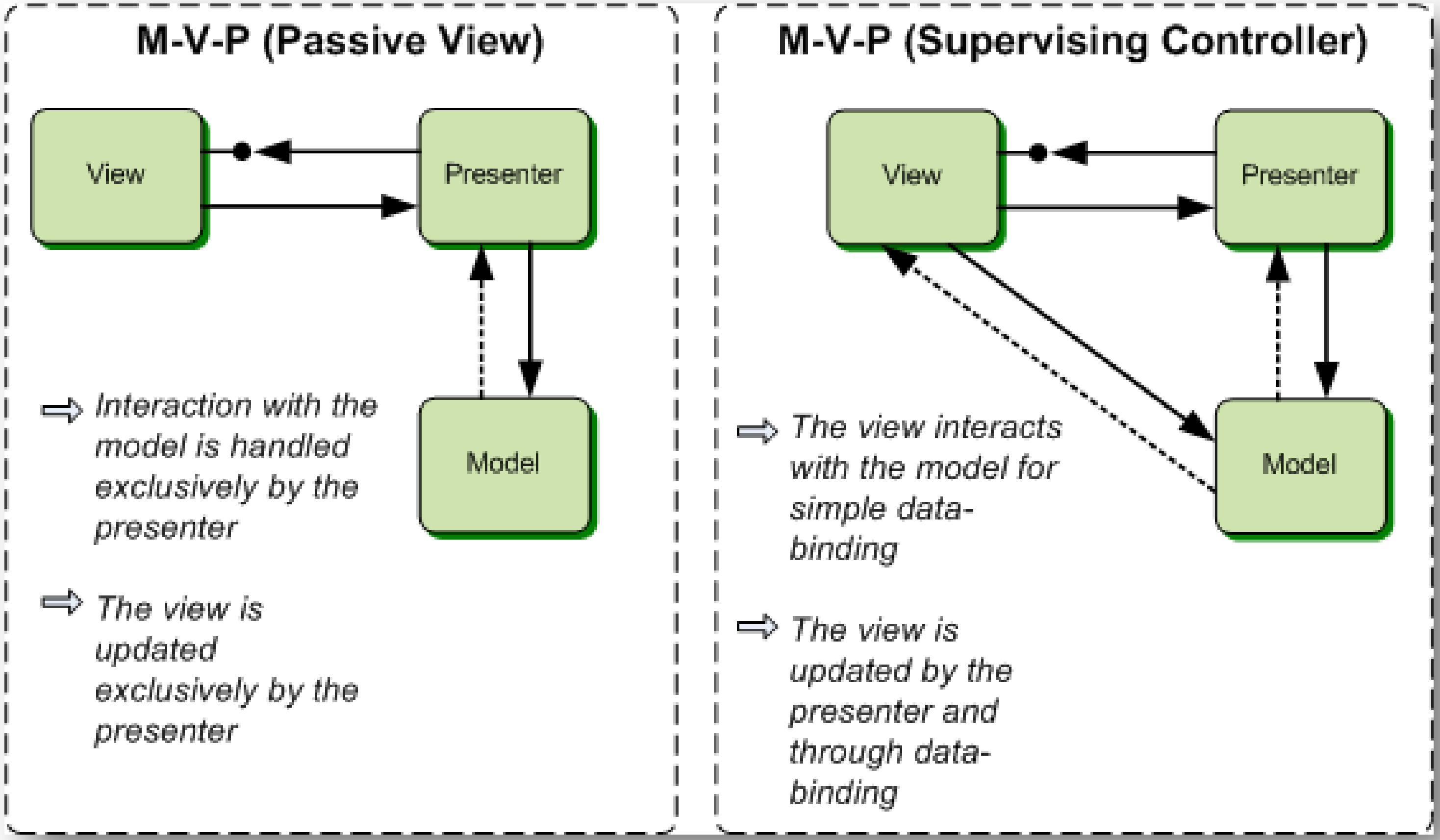
Figure 4: Simplistic MVC

MVP

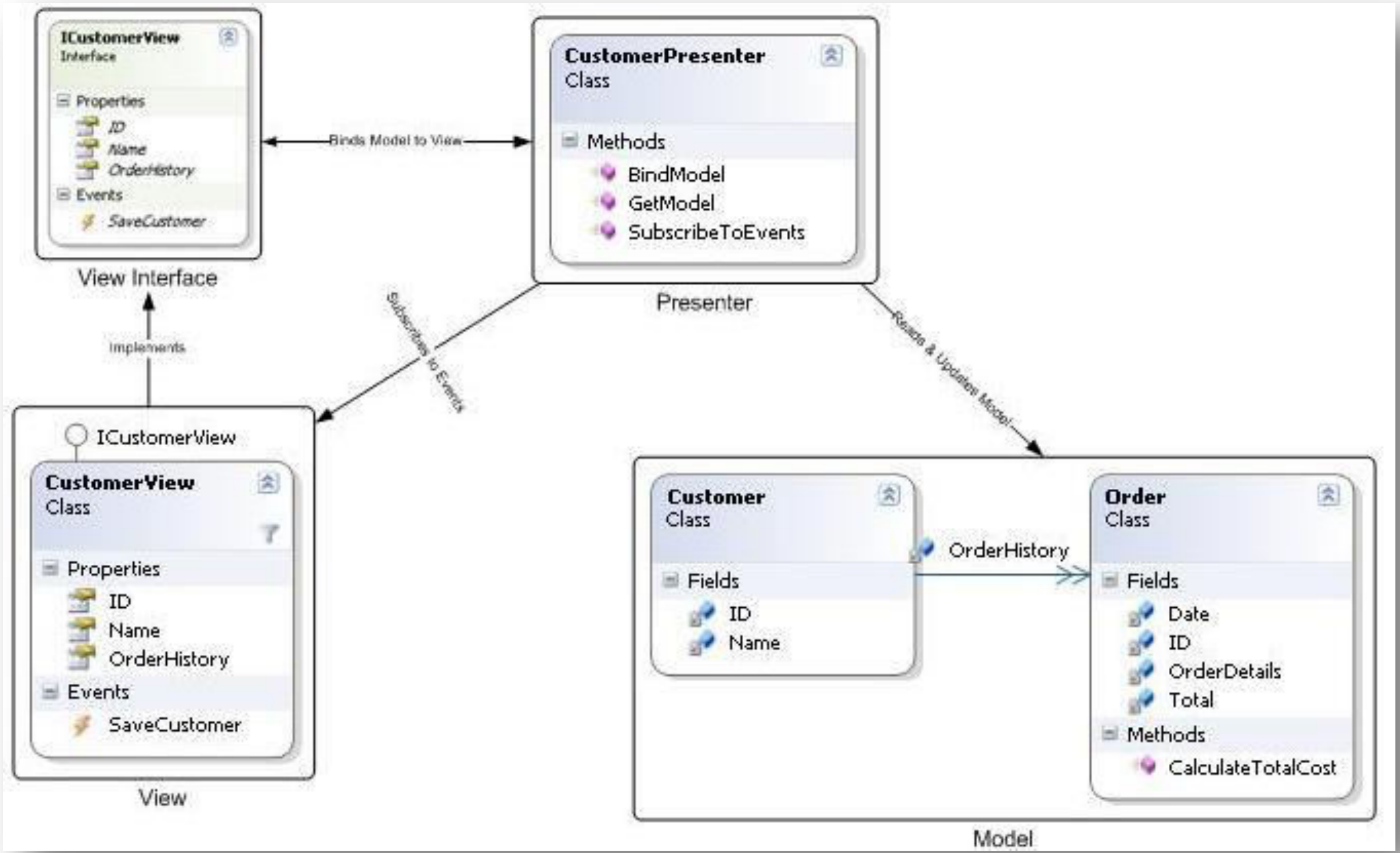
- The MVP pattern
  - is a UI presentation pattern based on the concepts of the MVC pattern.
- The pattern separates responsibilities across four components:
  - the view is responsible for rending UI elements,
  - the view interface is used to loosely couple the presenter from its view,
  - the presenter is responsible for interacting between the view/model,
  - 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.

View Updates

- When the model is updated, the view also has to be updated to reflect the changes.
- View updates can be handled in several ways.
  - Passive View – presenter does all the jobs
    - the presenter updates the view to reflect changes in the model.
    - presenter handlers the interaction with the model
    - the view is not aware of changes in the model.
  - Supervising Controller,
- In Supervising Controller,
  - the view interacts directly with the model to perform simple data-binding that can be defined declaratively, without presenter intervention.
  - The presenter updates the model; it manipulates the state of the view only in cases where complex UI logic that cannot be specified declaratively is required.
  - Examples of complex UI logic might include
    - changing the color of a control or dynamically hiding/showing controls.


M-V-P (Passive View)
View
Presenter
Model
Interaction with the model is handled exclusively by the presenter
The view is updated exclusively by the presenter
M-V-P (Supervising Controller)
View
Presenter
Model
The view interacts with the model for simple data-binding
The view is updated by the presenter and through data-binding

Example


ICustomerView
Interface
Properties
ID
Name
OrderHistory
Events
SaveCustomer
View Interface
Binds Model to View
CustomerPresenter
Class
Methods
BindModel
GetModel
SubscribeToEvents
Presenter
Implements
Subscribes to Events
Reads & Updates Model
ICustomerView
CustomerView
Class
Properties
ID
Name
OrderHistory
Events
SaveCustomer
View
Customer
Class
Fields
ID
Name
OrderHistory
Order
Class
Fields
Date
ID
OrderDetails
Total
Methods
CalculateTotalCost
Model

How to chose

- depends on how testable you want your application to be.
- If testability -> Passive View
- code simplicity -> Supervising Controller
- Other consideration
  - Both variants allow you to increase the testability of your presentation logic.
  - Passive View usually provides a larger testing surface than Supervising Controller because all the view update logic is placed in the presenter.
  - Supervising Controller typically requires less code than Passive View because the presenter does not perform simple view updates.