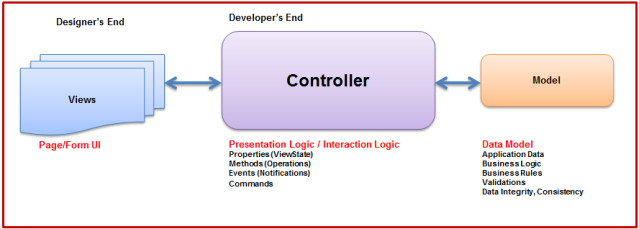
Coupling

Coupling is the dependency factor of one class on another that varies from low to high. In tightly coupled systems, changes in one class force the changes in another class and increase the efforts to make modifications. Tight coupling increases the maintenance cost and decreases reusability of component. So in order to achieve stable and low maintenance cost application, low degree of coupling is a desired trait of system design.

Loosely Coupled Application is to achieve separation of responsibilities and low coupling level. Architecture patterns are used to reduce coupling between objects, but architectural patterns help to improve system’s design.

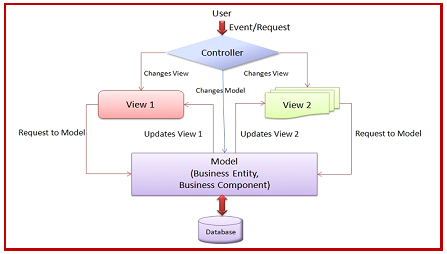
The main advantage of loosely coupling is its changeability without breaking other modules in an unpredictable manner.

* **Ease of Maintenance**: Loosely coupled application is able to change the implementation in a certain subsystem, or service without affecting another subsystem or service. It allows UI designers and developers to work in a self-governing manner. It means that if in future View (look and feel) replaces with another View, it should not require ViewModel and Model to change.
* **Ease of Testing**: Loosely coupled application provides the ability for ease of testability using the many available unit testing frameworks (Nunit, etc.).



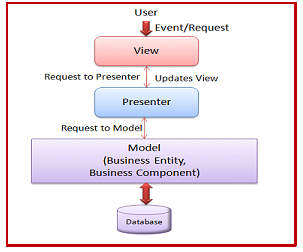
While designing complex UI applications, following are the four problems associated with UI:

* **State**: In Web Applications, session variables are used to maintain the state and in windows application it's simple UI level data. More efforts on state maintenance, the more it becomes complex.
* **Logic**: More logic to manipulate controls on User Interface, more complex it becomes.
* **Synchronization**: More the synchronization like coordination between UI controls and business objects, more complex becomes.
* **Caching**: Retrieving data from database for frequent access in application is the slowest way and a great hit on performance. So caching is the fundamental way of removing performance bottlenecks which results in slow data access. In case of single-tiered applications, different types of cache objects can be created at UI level (XAML, themes, images), business rules and Entity-Framework cache remain on the same page.
* **MVC (Model-View-Controller)**
* The main objective of MVC pattern is to decouple the view from the actual data processing so that the same model can be used for several views. This is accomplished by using three different objects (Model, View and Controller) that interact with each other in a loosely coupled manner.
* **Controller**: User interacts with the controller which in turn commands model and view to change. It’s responsible for Notice of Action.
* **View**: View is responsible for look and feel and isolated from the complex data processing. View can be XML
* **Model**: Model responds to requests made by the controller and notifies views to update their look and feel. In short, Model is display-neutral.



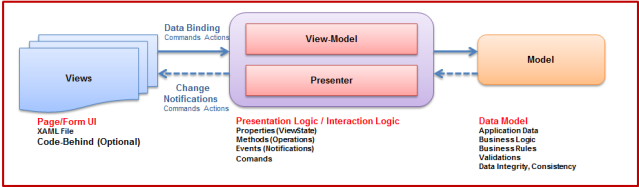
#### MVP (Model-View-Presenter)

A new approach was put forward by replacing Controller with Presenter and user requests are directed to View in spite of Presenter. Associated Presenter and View is having one-to-one mapping, i.e., the former is directly referenced with the View. Presenter updates the View (associated with) for the requested actions, which it performs on the Model.



#### MVVM (Model-View-ViewModel)

The Model-View-ViewModel (MVVM) architecture pattern approach separates the business logic and presentation logic of the application, from user interface (UI). By maintaining a separation between the application logic and user interface (UI), it helps to address design and development issues which can result in making the application much easier to test and maintain. It can also greatly improve code reusability that allows UI designers and developers to more easily collaborate when developing their respective parts of the application.



MVVM design pattern, the UI of the application and the underlying presentation and business logic can be separated into three separate classes: the view, which encapsulates the UI and UI logic; the view model, which encapsulates presentation logic; and the model, which encapsulates the application's business logic and data.

##### The View Class

The View class’s responsibility is to define the structure and lookup that user views on screen. It is a visual element, such as a page, user control, window, or data template. It uses the dominant data-binding capabilities of Silverlight/WPF to bind the properties defined in ViewModel to user controls that View is composed of.

User-events that are captured by the View are sent to the ViewModel through Commands. Now these commands execute methods defined in the ViewModel which contains logic. To update the View with updated data in viewmodel’s properties keeps the View bound to it.

As per MVVM compliant application, the View should contain minimum code, in an exceptional case code for a specific view. It contains UI controls and bindings of the properties defined in viewmodel to the controls which is again specific to the View. Code for validations specific to the View is also mentioned in the View. View has many-to-one relationship towards ViewModel. For reusability of code, User Control is used and data context of the ViewModel is set in view. Styles are also provided in the View only. In short, View is independent of viewmodel and vice-versa is also true.

##### The Model Class

Model Class is responsible for data representation. It inherits INotifyPropertyChangedinterface to notify ViewModel if properties defined in Model are modified. It is used to expose the data objects in an appropriate way that those can be easily managed and consumed by View and ViewModel. Objects like IDataErrorInfoare inherited in Model class to provide validations for the properties defined. RaisePropertyChanged method is also added in Model for every property onPropertyChangedEventHandler.

##### The ViewModel Class

ViewModel in the MVVM design pattern encapsulates the presentation logic and data for the view. It simply holds the data retrieved from data layer and View holds the formatted information, and ViewModel acts like a liaison in between the two. It is never tightly coupled with the View and notifies later, if any property changes in ViewModel. It might take inputs from View and place it on Model, or it might interact with a service to retrieve the model, then translate properties and place it on View. It also exposes methods, commands, and helps to maintain the state of the View, manipulates the Model as the result of actions on the View, and triggers events in View itself. ViewModel always updates Model and properties defined from View having UI level events mapped to commands in ViewModel by two-way data binding.