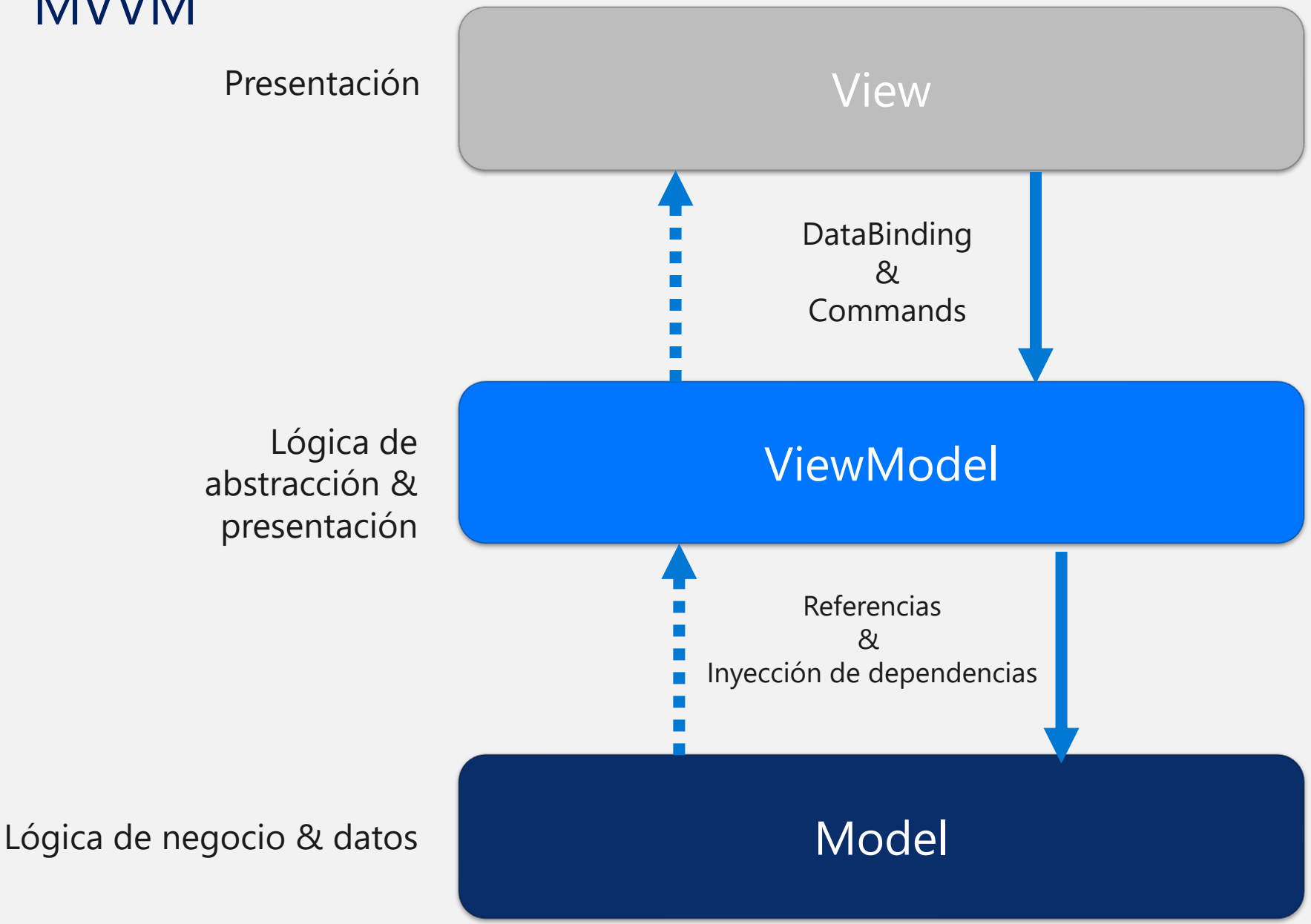
53. MVVM

MVVM


Presentación
View
DataBinding
&
Commands
Lógica de
abstracción &
presentación
ViewModel
Referencias
&
Inyección de dependencias
Lógica de negocio & datos
Model

¿Qué es el Model?

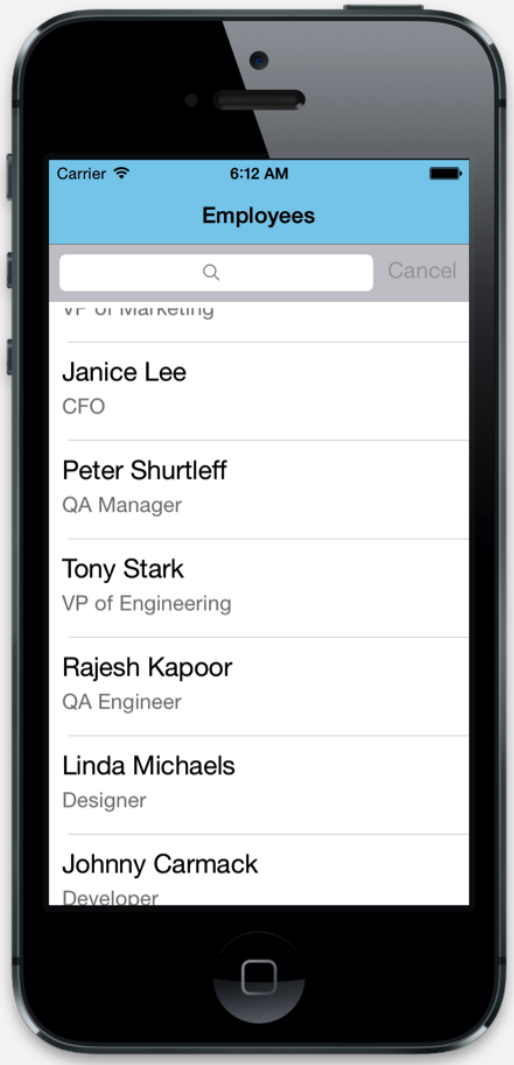
Los modelos administran los **datos** de la aplicación y pueden incluir cualquier combinación de lógica de dominio, estado persistente y validación, no necesariamente en un objeto.

Los modelos están destinados a ser **compartidos entre plataformas** y no deben depender de las características específicas de la plataforma.

```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Title { get; set; }
    public DateTime HireDate { get; set; }
    public int Supervisor { get; set; }
    public static Employee GetById(int id);
    public static void UpdateRecord(Employee employee);
}
```

¿Qué es la View?

- La View presenta la información al usuario de una manera específica de la plataforma
- No debe contener el código que desea probar unitariamente
- Todo lo visual debe administrarse aquí: fuentes, colores, etc.



¿Qué es la ViewModel?

La ViewModel proporciona una representación de los datos para mostrar en la View

expone propiedades enlazables e implementa notificación de cambio de propiedad

```
public class EmployeeViewModel : INotifyPropertyChanged
{
    private Employee model;

    public string Name {
        get { return model.Name; }
        set { model.Name = value; OnPropertyChanged("Name"); }
    }

    public EmployeeViewModel(Employee model) {
        model = model;
    }
    ...
}
```

a menudo tiene una relación 1:1 con el modelo

¿Qué es la ViewModel?

... permite una conversión/coerción de métodos o propiedades del modelo

```
partial class EmployeeViewModel
{
    ...
    public string DateHiredText {
        get { return model.HireDate.ToString("MMM d, yyyy"); }
    }

    public EmployeeViewModel Supervisor {
        get { return new EmployeeViewModel(
                                Employee.GetById(this.supervisor)); }
    }
}
```

¿Qué es la ViewModel?

- … proporciona una forma enlazable de acceder a datos relacionados

```
partial class EmployeeViewModel
{
    ...
    public IEnumerable<string> ActiveProjects {
       get {
           return CompanyProjects.All
             .Where(p => p.Owner == model.Id
                    && p.IsActive)
             .Select(p => p.Name).ToList();
        }
    }
}
```

¿Qué es la ViewModel?

… y proporciona un lugar conveniente para poner lógica necesaria para la interfaz de usuario

- Realice la validación de entrada antes de almacenarla en el modelo
- Realice cálculos visuales o valores de estado de tiempo de ejecución para la interfaz de usuario

```
partial class DownloaderViewModel {
   private int percentComplete;
   public int PercentComplete {
      get { return percentComplete; }
      set {
         if (percentComplete != value) {
            percentComplete = value;
            RaisePropertyChanged("PercentComplete");
         }
      }
   }
}
```

Puede usar la propiedad para impulsar una notificación de progreso en la interfaz de usuario

Conectando la Vista con la ViewModel

La ViewModel se establece con frecuencia como BindingContext para la vista en el código subyacente, pero también se puede hacer en XAML si se prefiere.

```
public partial class MainPage : ContentPage
{
    readonly MainViewModel viewModel = new MainViewModel();
    public MainPage ()
    {
        BindingContext = viewModel;
        InitializeComponent ();
    }
    ...
}
```

Pros & Cons

MVVM Pros and Cons

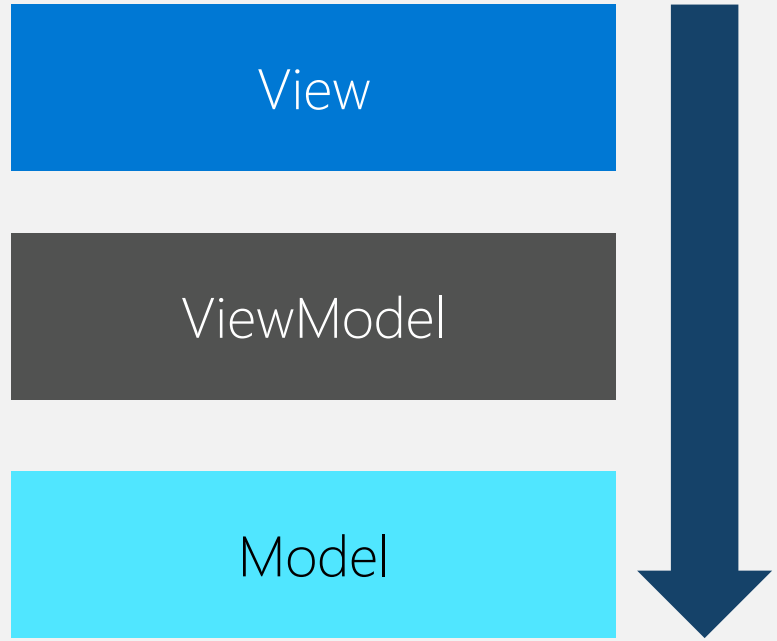
MVVM es ideal para plataformas con una infraestructura de enlace de datos como .NET MAUI y es la arquitectura preferida para aplicaciones no triviales.

Pros	Cons
▪ Facilita la capacidad de testing ▪ Centraliza la lógica de negocios ▪ Puede reducir el código necesario utilizado para vincular modelos a la interfaz de usuario ▪ Aprovecha la infraestructura de Bindings	▪ Requiere infraestructura ▪ Los enlaces pueden ser difíciles de depurar y pueden no ser eficientes para grandes conjuntos de datos

La ViewModel

View vs. ViewModel

- La ViewModel está vinculado intencionalmente a la vista, pero debe escribirse para que sea independiente de la interfaz de usuario
  - por lo tanto, no debe tener dependencias de nada relacionado con .NET MAUI



Cada capa solo debe tener conocimiento directo sobre la capa debajo de ella

Trabajando con propiedades relacionadas con la UI

Suponga que un requisito comercial es cambiar el color del nombre del empleado en la interfaz de usuario si es un supervisor

```
partial class EmployeeViewModel
{
    public Color NameColor { get; }
}
```

¡Evita esto! El color es un tipo específico de .NET MAUI

... esto es mejor pero aún no es ideal: los colores deben ser determinados por el rol del diseñador y el código de vista

```
partial class EmployeeViewModel
{
    public string NameColor { get; }
}
```

Trabajando con propiedades relacionadas con la UI

Suponga que un requisito comercial es cambiar el color del nombre del empleado en la interfaz de usuario si es un supervisor

```
partial class
{
    public Co
}
```

```
partial class EmployeeViewModel
{
    public bool IsSupervisor {
        get { ... }
        private set {
    }
}
```

Expongamos una propiedad booleana que indique si el empleado tiene subordinados...

... esto es mejor pero aún no es ideal: los colores deben ser determinados por el rol del diseñador y el código de vista

```
del

    public string NameColor { get; }
}
```

or es un tipo
T MAUI

Trabajando con propiedades relacionadas con la UI

Los activadores de datos admiten cambios dinámicos en las propiedades de la interfaz de usuario basados en enlaces con condicionales

```
<Label Text="{Binding Name}" TextColor="Gray">
    <Label.Triggers>
        <DataTrigger TargetType="Label"
                     Binding="{Binding IsSupervisor}"
                     Value="True">
            <Setter Property="TextColor" Value="Blue" />
        </DataTrigger>
    </Label.Triggers>
</Label>
```

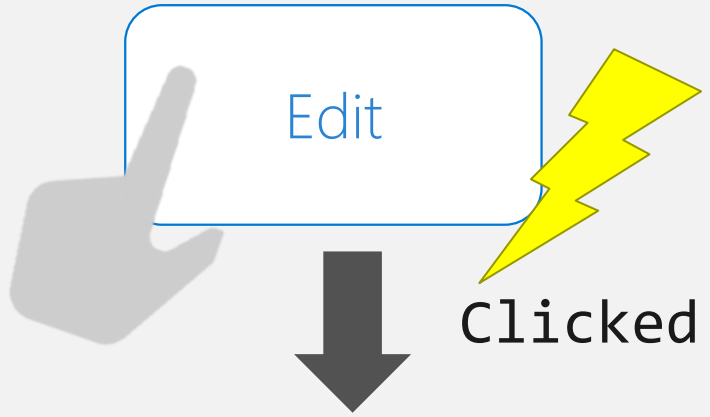
Comandos

Gestión de eventos

La interfaz de usuario genera eventos para notificar al código sobre la actividad del usuario

- Hizo clic
- Elemento seleccionado…

La desventaja es que estos eventos deben manejarse en el archivo de código subyacente



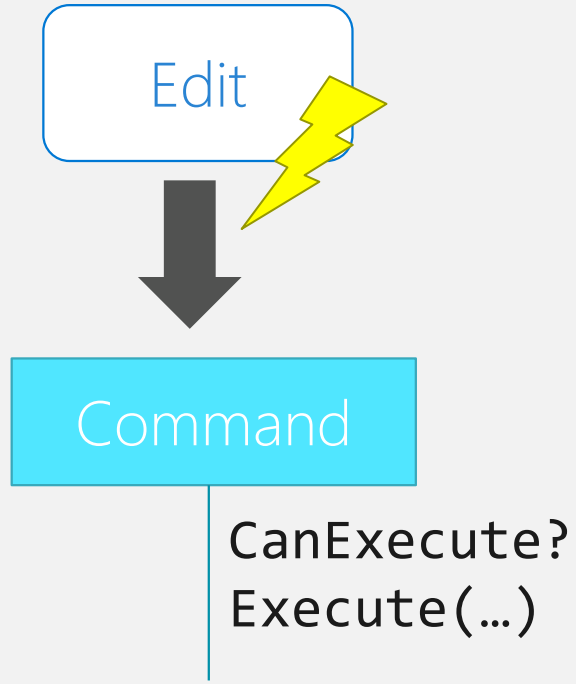

```
public MainPage()
{
    ...
    Button editButton = ...;
    editButton.Clicked += OnClick;
}

void OnClick (object sender, EventArgs e)
{
    ...
}
```

Commands

Microsoft definió la interfaz ICommand para proporcionar una abstracción dominante para sus frameworks XAML

```
public interface ICommand
{
    bool CanExecute(object parameter);
    void Execute(object parameter);
    event EventHandler CanExecuteChanged;
}
```

Puede proporcionar un parámetro opcional (a menudo null) para que el comando funcione para el contexto



Commands en .NET MAUI

Algunos controles de .NET MAUI exponen una propiedad **Command** para la acción principal de un control.

`Button`

`Menu`

`ToolbarItem`

`TextCell`

Commands en .NET MAUI

Algunos controles de .NET MAUI exponen una propiedad **Command** para la acción principal de un control.

```
public ICommand GiveBonus { get; }
```

```
<Button Text="Give Bonus"
        Command="{Binding GiveBonus}" />
```

Puedes vincular datos de una propiedad de tipo ICommand a la propiedad Command

Gesture-based commands

- Xamarin.Forms also includes a `TapGestureRecognizer` which can provide a command interaction for other controls or visuals

```
<Image Source="IDareYouToTapMe.jpg">
    <Image.GestureRecognizers>
        <TapGestureRecognizer
            Command="{Binding BeBraveCommand}"
            CommandParameter="TheyTookTheDare!" />
    </Image.GestureRecognizers>
</Image>
```

`CommandParameter` property supplies the command's parameter – in this case as a `string`

Implementación del comando en la ViewModel

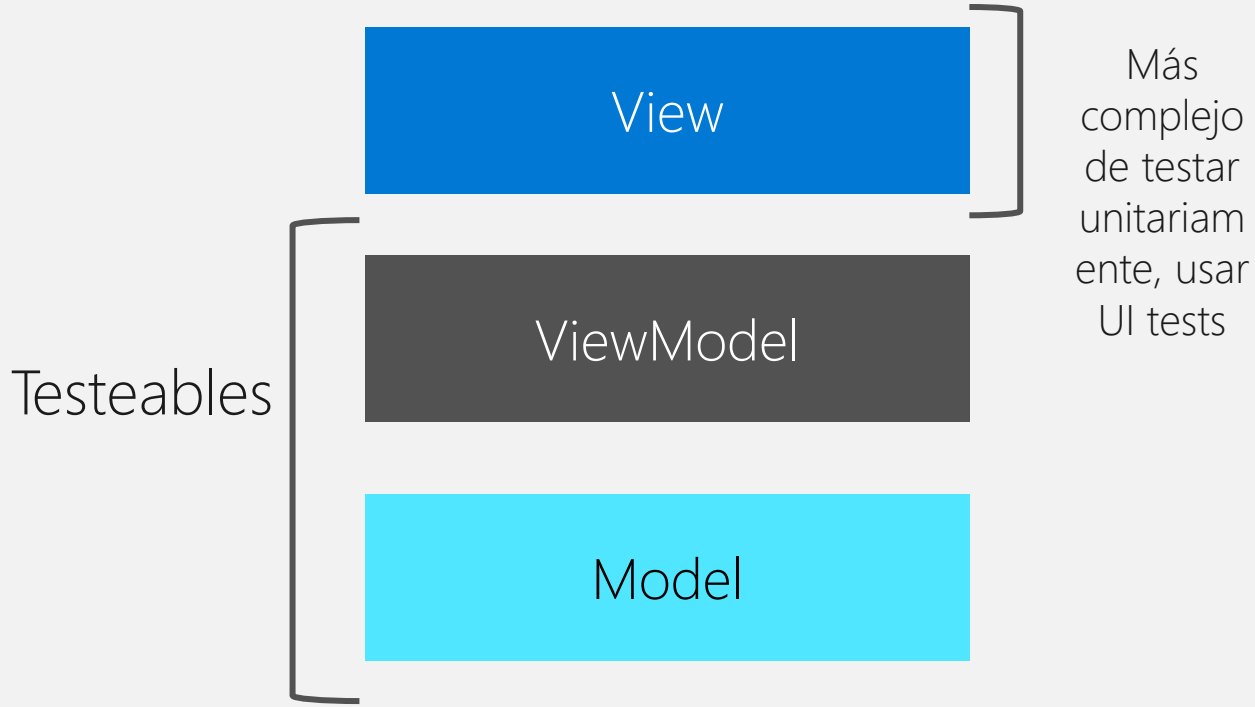
El comando debe exponerse como una propiedad pública de ViewModel

```
public class EmployeeViewModel : INotifyPropertyChanged
{
    public ICommand GiveBonus { get; private set; }
    ...
    public EmployeeViewModel(Employee model) {
        this.model = model;
        GiveBonus = new Command(ExecuteGiveBonus);
    }
    ...
}
```

Testing

Testing

- Las pruebas unitarias implican probar piezas pequeñas y aisladas de nuestra aplicación de forma independiente; eso es muy difícil de hacer para aplicaciones GUI estrechamente acopladas
- El código comprobable es código que no tiene dependencias en la presencia de una interfaz de usuario



Testing de la ViewModel

- La ViewModel se puede probar independientemente de la interfaz de usuario/plataforma.
- Permite probar la lógica empresarial y la lógica visual.
- Puede usar frameworks de testing unitario bien conocidos como NUnit o xUnit.

Testing de la ViewModel

set properties and invoke command – just like UI would

```
[Fact]
void Employee_GiveBonus_Succeeds()
{
    var data = new Employee(...);
    var vm = new EmployeeViewModel(data);
    vm.GiveBonus.Execute("500");

    Assert.Equal(500,
            data.GetNextPaycheckData().Extras);
}
```

… and then test the results to verify it does what you expect