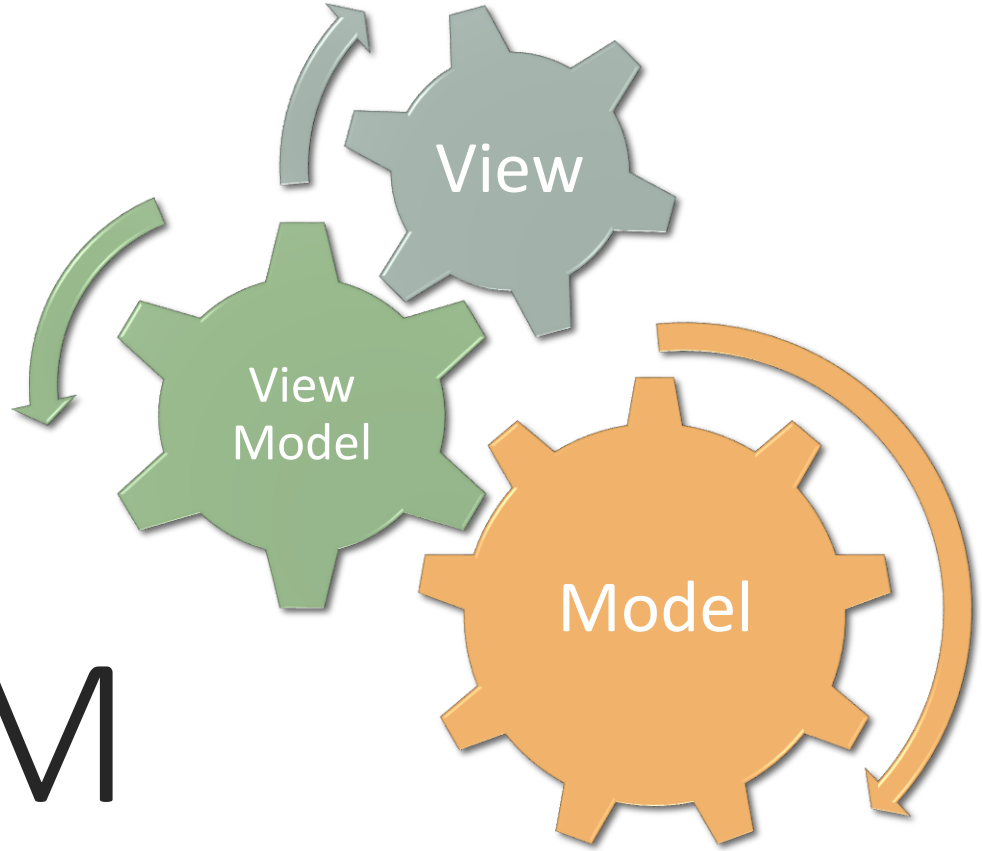
Advanced Programming 2

DR. ELIAHU KHALASTCHI

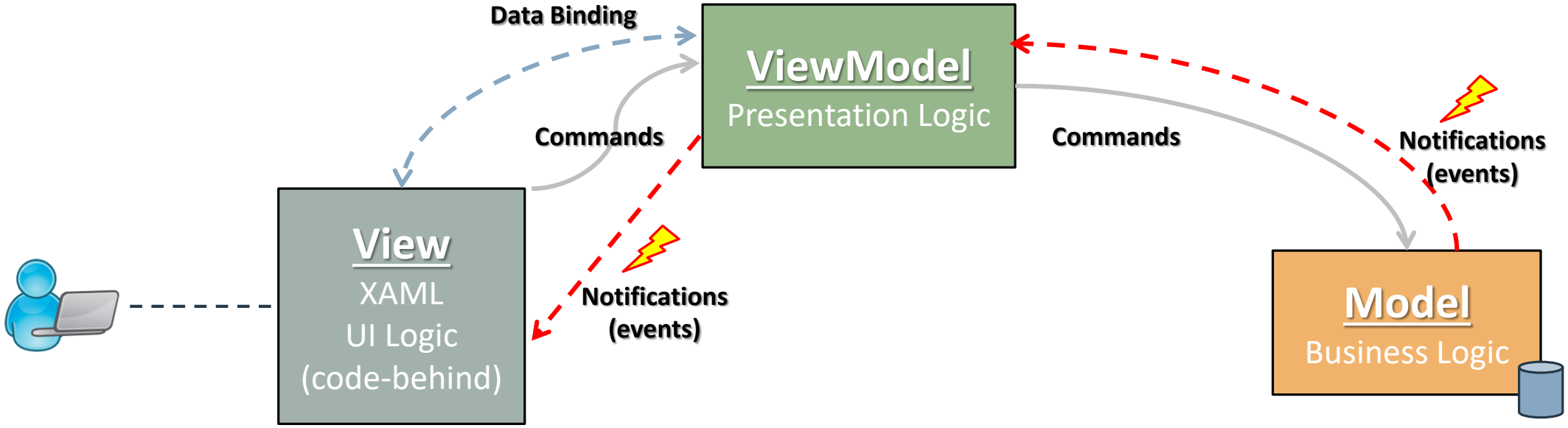
2016



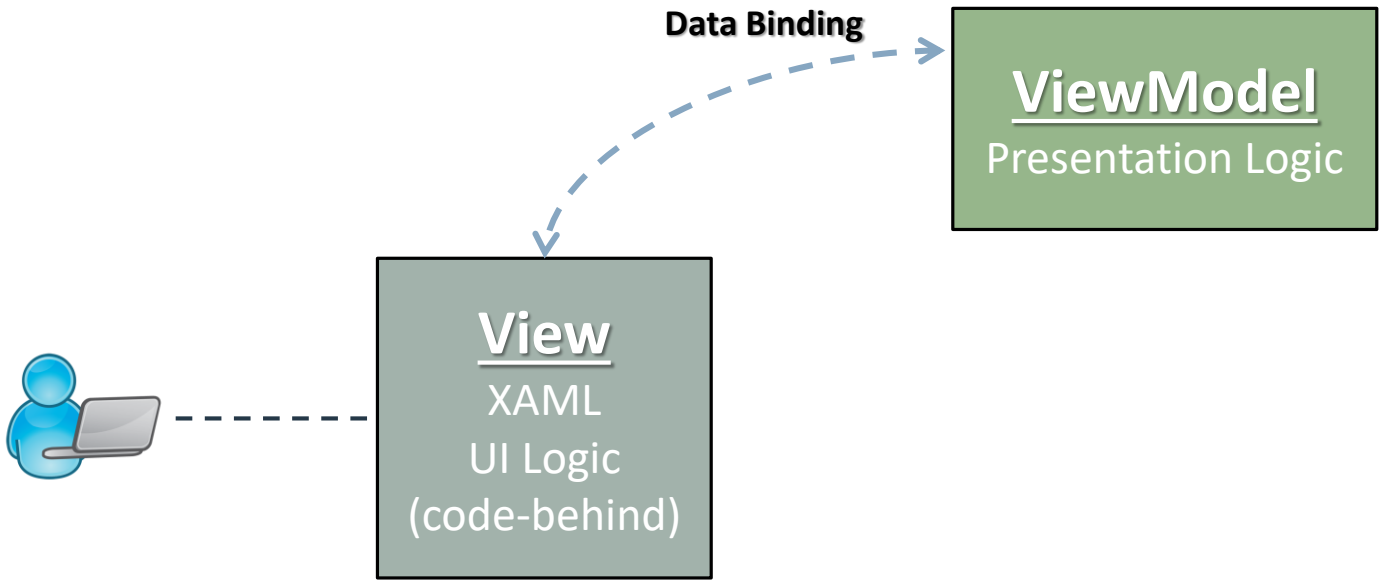
MVVM

MODEL, VIEW, VIEW MODEL

The MVVM Architecture



- The **ViewModel** is the "model of the view"
  - For the View, it is an abstraction of the Model
  - It passes commands from the view to the model
- The **ViewModel**
  - Converts model information into view information
  - Something the View can understand…



Data Biding

ELIAHU KHALASTCHI

Source: http://www.wpf-tutorial.com/data-binding/

Agenda

- Data binding within the UI
- Data Binding with other objects
- Observable Collection
- INotifyPropertyChanged interface

Introduction

- Data binding is general technique that binds two data/information sources together
- and maintains synchronization of data

- Data binding in WPF is the **preferred way** to bring data from your code to the UI layer

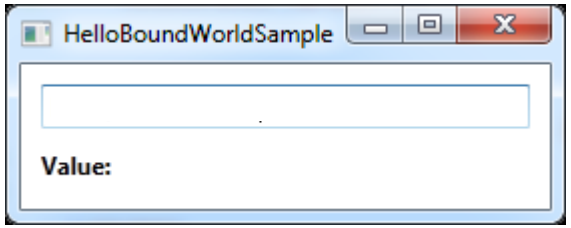
- Sure, you can set properties on a control manually
- but the cleanest and purest WPF way is to add a binding between
  - the source
  - and the destination UI element

The Syntax of Binding

- The TextBlock matches the TextBox upon typing!

- We **didn't** have to use code behind:
  - Listen to a TextBox event
  - Update the TextBlock with each change
- We used only markup!

```
<StackPanel Margin="10">
        <TextBox Name="txtValue" />
        <WrapPanel Margin="0,10">
            <TextBlock Text="Value: " FontWeight="Bold" />
            <TextBlock Text="{Binding Path=Text, ElementName=txtValue}" />
        </WrapPanel>
</StackPanel>
```



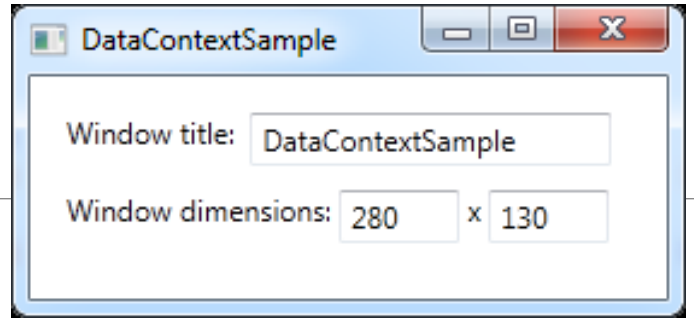
- Syntax:
  - **Some_Property** = {Binding Path=**the_Property_of**, ElementName = **another_Element**}
  - The Path property is a default, we can just write Text={Binding Text, ElementName=txtValue}

Data Context

- The DataContext property is the **default source** of your bindings
  - unless you specifically declare another source,
  - like we did in the previous example

- A DataContext is **inherited** down through the control hierarchy
- For example, you can set a DataContext for the Window itself
- Then use it throughout all of the child controls

Data Context Example




```
<StackPanel Margin="15">
        <WrapPanel>
            <TextBlock Text="Window title:  " />
            <TextBox Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}" Width="150" />
        </WrapPanel>
        <WrapPanel Margin="0,10,0,0">
            <TextBlock Text="Window dimensions: " />
            <TextBox Text="{Binding Width}" Width="50" />
            <TextBlock Text=" x " />
            <TextBox Text="{Binding Height}" Width="50" />
        </WrapPanel>
</StackPanel>
```

```
public partial class DataContextSample : Window {
        public DataContextSample() {
            InitializeComponent();
            this.DataContext = this;
        }
}
```

The title changes immediately but the window's dimensions changes only after the text box lost it's focus…

The default is:

```
<TextBox Text="{Binding Height, UpdateSourceTrigger=LostFocus }" Width="50" />
```

So this would do the trick:

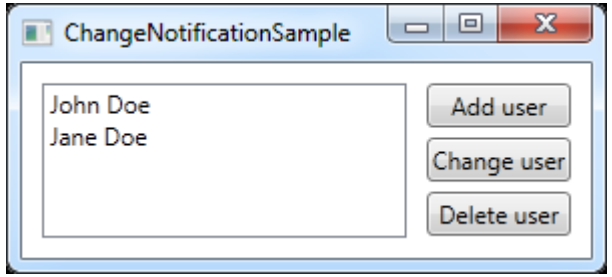
```
<TextBox Text="{Binding Height, UpdateSourceTrigger=PropertyChanged }" Width="50" />
```

Real world binding example

- Consider we have a simple class called **User** with a **Name** property
- We may also have a list of users and whish to display it in a list box

```
public class User
{
   public string Name { get; set; }
}
```

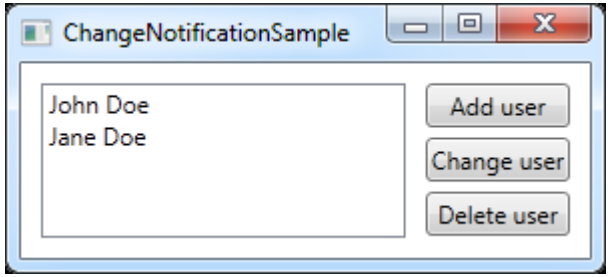
```
<DockPanel Margin="10">
  <StackPanel DockPanel.Dock="Right" Margin="10,0,0,0">
     <Button Name="btnAddUser" Click="btnAddUser_Click">Add user</Button>
     <Button Name="btnChangeUser" Click="btnChangeUser_Click" Margin="0,5">Change user</Button>
     <Button Name="btnDeleteUser" Click="btnDeleteUser_Click">Delete user</Button>
  </StackPanel>
  <ListBox Name="lbUsers" DisplayMemberPath="Name"></ListBox>
</DockPanel>
```



The code behind…

```
public partial class ChangeNotificationSample : Window {
        private List<User> users = new List<User>();

        public ChangeNotificationSample() {
            InitializeComponent();

            users.Add(new User() { Name = "John Doe" });
            users.Add(new User() { Name = "Jane Doe" });

            lbUsers.ItemsSource = users;
        }

        private void btnAddUser_Click(object sender, RoutedEventArgs e) {
            users.Add(new User() { Name = "New user" });
        }

        private void btnChangeUser_Click(object sender, RoutedEventArgs e) {
            if (lbUsers.SelectedItem != null)
                (lbUsers.SelectedItem as User).Name = "Random Name";
        }

        private void btnDeleteUser_Click(object sender, RoutedEventArgs e) {
            if (lbUsers.SelectedItem != null)
                users.Remove(lbUsers.SelectedItem as User);
        }
    }
```

But this does not work….



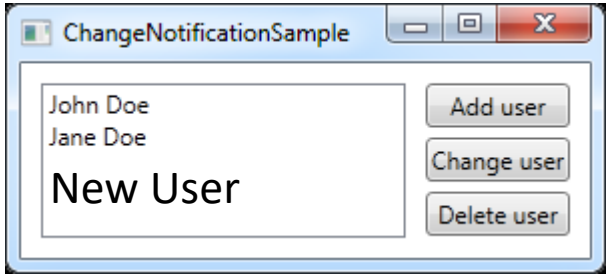
Observable Collection

- To make the Add & Delete buttons to work we simply replace the ***List<User>*** with ***ObservableCollection<User>***
- When the ObservableCollection is changed it notifies automatically its observers
- In our case, the *ListBox* is observing the users list
  - Because: `lbUsers.ItemsSource = users;`
- Adding or deleting users from *users* will cause the *ListBox* to reflect these changes
- But what about the change user button?

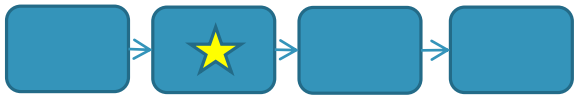
The code behind…

```
public partial class ChangeNotificationSample : Window {
    private List<User> users = new List<User>();
    private ObservableCollection <User> users = new ObservableCollection <User>();

      public ChangeNotificationSample() {
          InitializeComponent();

          users.Add(new User() { Name = "John Doe" });
          users.Add(new User() { Name = "Jane Doe" });

          lbUsers.ItemsSource = users;
      }

      private void btnAddUser_Click(object sender, RoutedEventArgs e) {
          users.Add(new User() { Name = "New user" });
      }
      private void btnChangeUser_Click(object sender, RoutedEventArgs e) {
          if (lbUsers.SelectedItem != null)
              (lbUsers.SelectedItem as User).Name = "Random Name";
      }
      private void btnDeleteUser_Click(object sender, RoutedEventArgs e) {
          if (lbUsers.SelectedItem != null)
              users.Remove(lbUsers.SelectedItem as User);
      }
  }
```

Change User does not work….



users:



The INotifyPropertyChanged interface

```
public class User : INotifyPropertyChanged {

    private string name;
    public string Name {
        get { return this.name; }
        set {
            if (this.name != value {
                this.name = value;
                this.NotifyPropertyChanged("Name");
            }
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    public void NotifyPropertyChanged(string propName){
        if (this.PropertyChanged != null)
            this.PropertyChanged(this, new PropertyChangedEventArgs(propName));
    }
}
```

```
interface INotifyPropertyChanged {
    event PropertyChangedEventHandler PropertyChanged;
}
```

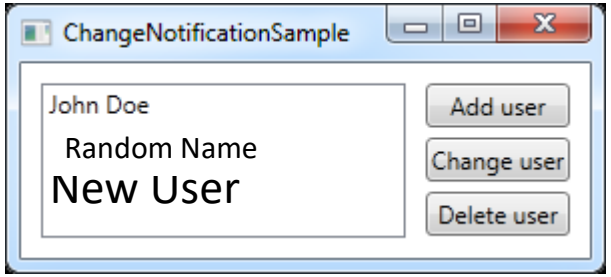
```
public delegate void PropertyChangedEventHandler(
   Object sender,
   PropertyChangedEventArgs e
)
```

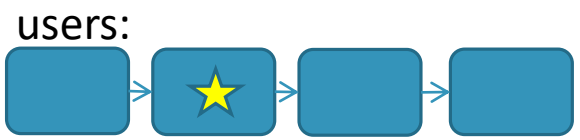
The sender

the property-changed event arguments,
contains the name of the property that has changed

The code behind…

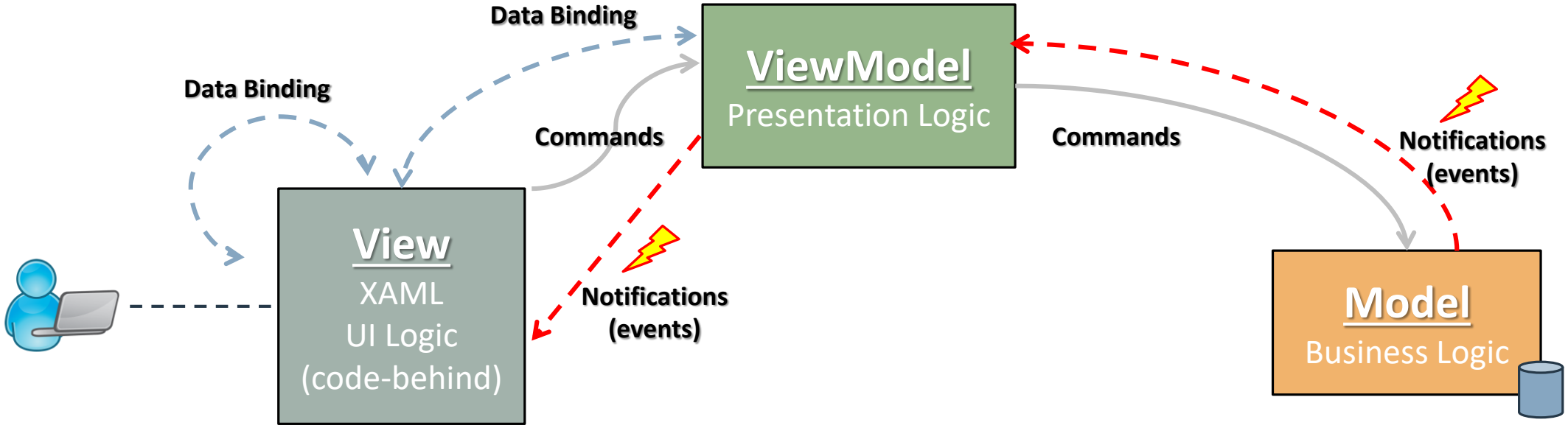
```
public partial class ChangeNotificationSample : Window {
    private List<User> users = new List<User>();
    private ObservableCollection <User> users = new ObservableCollection <User>();

      public ChangeNotificationSample() {
          InitializeComponent();

          users.Add(new User() { Name = "John Doe" });
          users.Add(new User() { Name = "Jane Doe" });

          lbUsers.ItemsSource = users;
      }

      private void btnAddUser_Click(object sender, RoutedEventArgs e) {
          users.Add(new User() { Name = "New user" });
      }
      private void btnChangeUser_Click(object sender, RoutedEventArgs e) {
          if (lbUsers.SelectedItem != null)
              (lbUsers.SelectedItem as User).Name = "Random Name";
      }
      private void btnDeleteUser_Click(object sender, RoutedEventArgs e) {
          if (lbUsers.SelectedItem != null)
              users.Remove(lbUsers.SelectedItem as User);
      }
  }
```



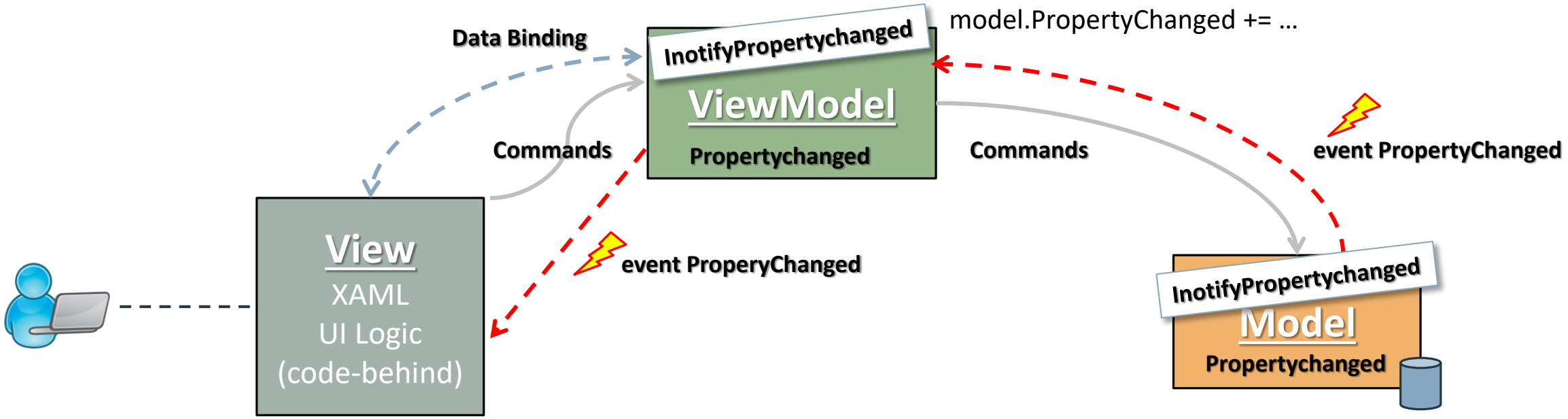


Where “users” should actually be?

The MVVM Architecture

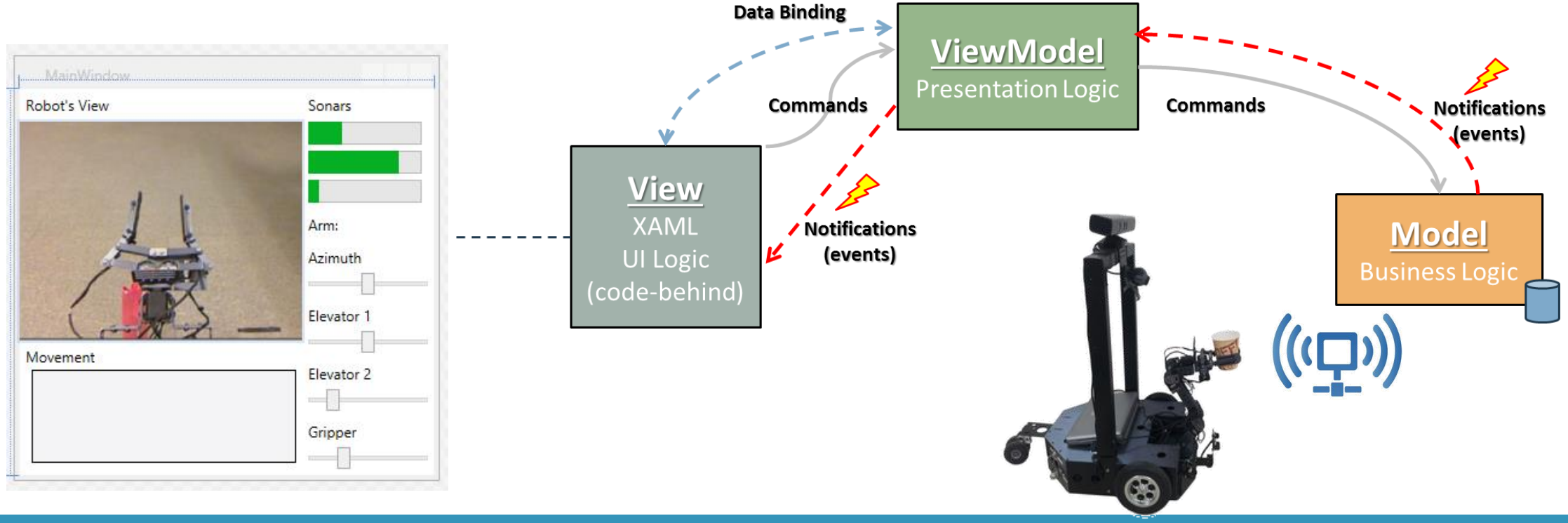


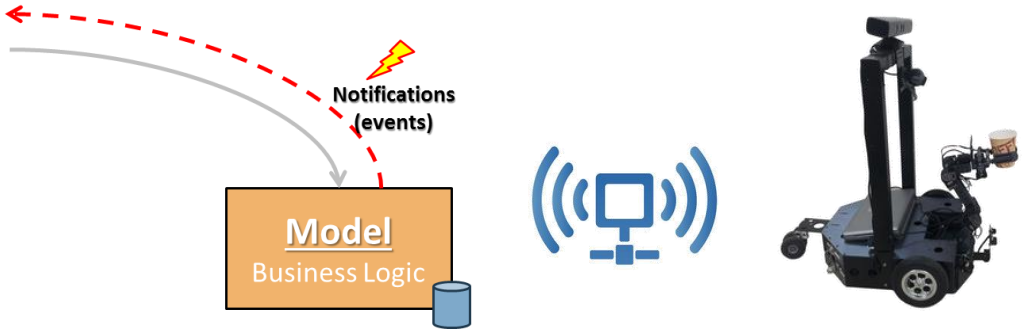
The MVVM Architecture



Robot Remote Control MVVM Example

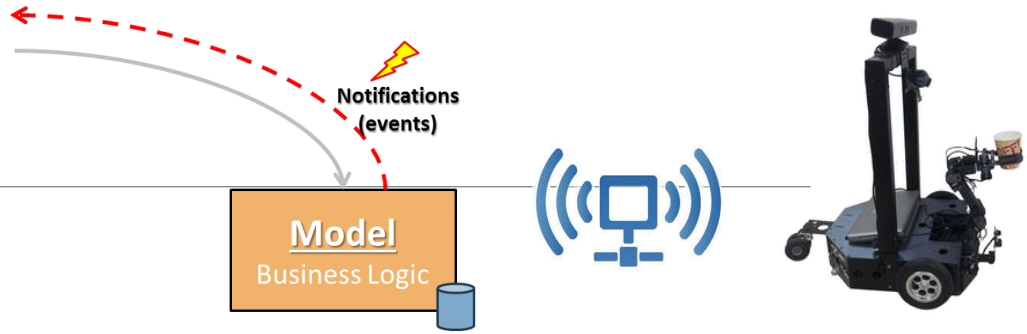
- We want to build a GUI to remotely control a robot





The Model

The Model

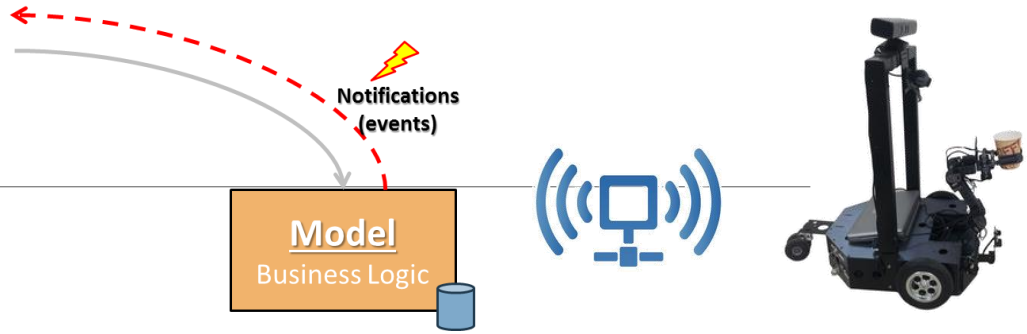


- Implements INotifyPropertyChanged

```
interface IRobotModel : INotifyPropertyChanged {
     // connection to the robot
     void connect(string ip, int port);
     void disconnect();
     void start();

     // sensors properties
     double LeftSonar { set; get; }
     double CenterSonar { set; get; }
     double RightSonar { set; get; }
     byte[][] CamView { set; get; }

     // activate actuators
     void move(double speed, int angle);
     void moveArm(int az, int e1, int e2, bool grip);
}
```

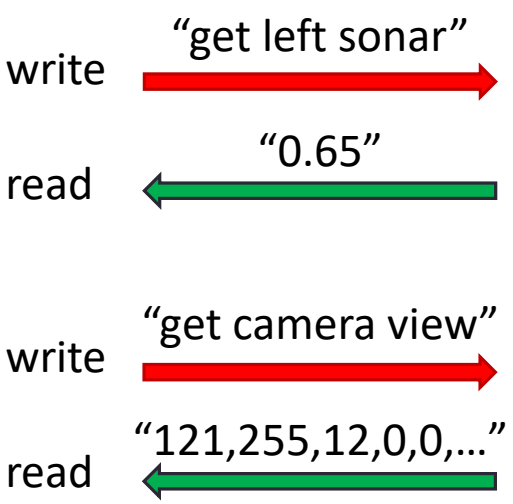
```
class MyRobotModel : IRobotModel {

   //INotifyPropertyChanged implementation:
   public event PropertyChangedEventHandler PropertyChanged;

   // the properties implementation
   public double LeftSonar{
       get { throw new NotImplementedException(); }
       set { throw new NotImplementedException(); }
   }

   // the rest of the properties...

   // the rest of IRobotModel's methods...
```

The Model



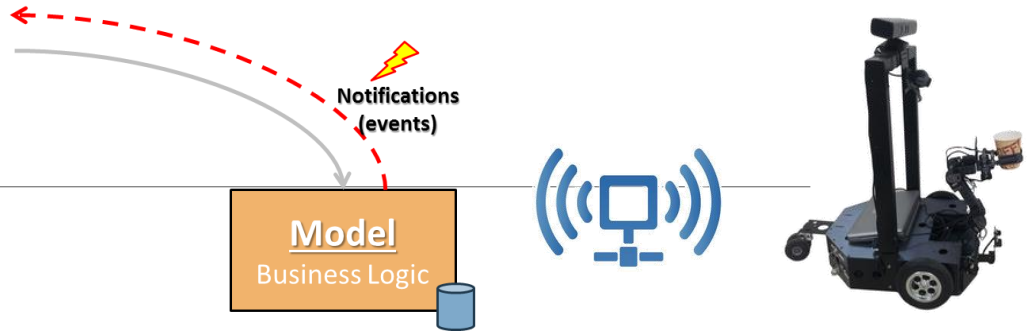
- Uses a ***telnet client*** for the communication with the robot

```
interface ITelnetClient {
        void connect(string ip, int port);
        void write(string command);
        string read();  // blocking call
        void disconnect();
}

class MyTelnetClient : ITelnetClient {...
```



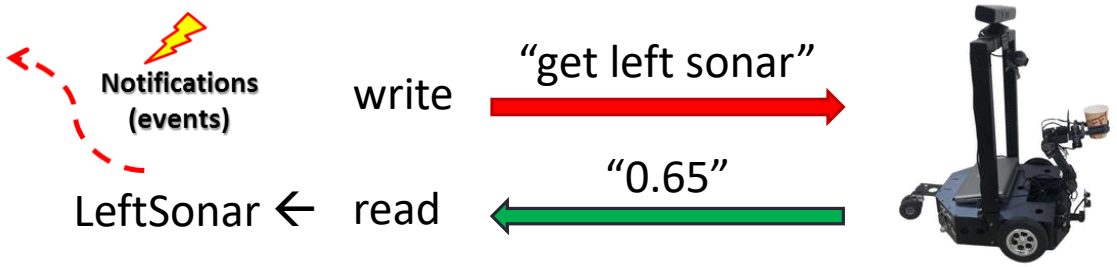
The Model

- Uses a ***telnet client*** for the communication with the robot

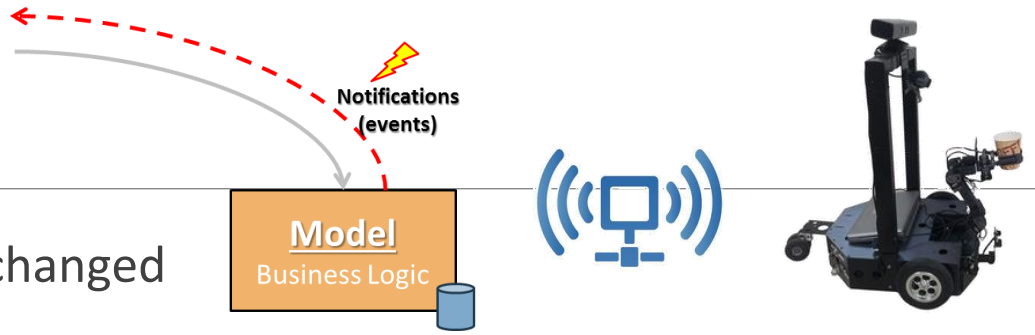



```
class MyRobotModel : IRobotModel {

    ITelnetClient telnetClient;
    volatile Boolean stop;

    public MyRobotModel(ITelnetClient telnetClient){
        this.telnetClient = telnetClient;
        stop = false;
    }

    public void connect(string ip, int port) {
        telnetClient.connect(ip, port);
    }

    public void disconnect(){
        stop = true;
        telnetClient.disconnect();
    }
```

```
public void start(){
    new Thread(delegate(){
        while (!stop)
        {
            telnetClient.write("get left sonar");
            LeftSonar = Double.Parse(telnetClient.read());
            // the same for the other sensors properties

            Thread.Sleep(250);// read the data in 4Hz
        }
    }).Start();
}
```



The Model

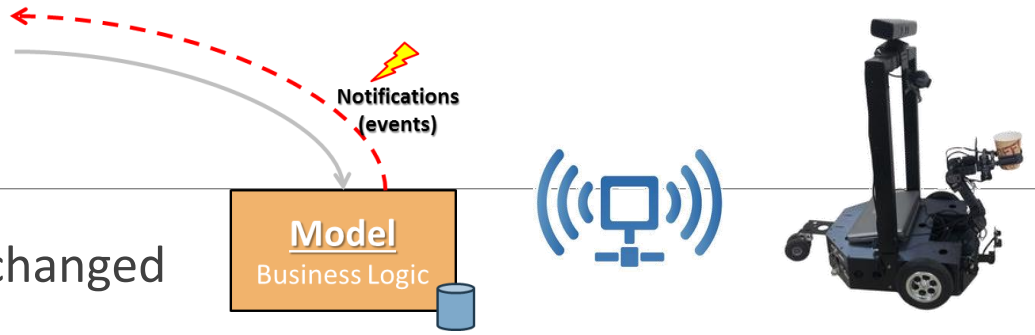


- sending an event that a property has changed

```
// inside MyRobotModel
public event PropertyChangedEventHandler PropertyChanged;

private double leftSonar;
public double LeftSonar
{
     get { return leftSonar; }
     set {
          leftSonar = value;
          NotifyPropertyChanged("LeftSonar");
     }
}

public void NotifyPropertyChanged(string propName){
     if (this.PropertyChanged != null)
          this.PropertyChanged(this, new PropertyChangedEventArgs(propName));
}
```

```
public void start(){
   //... In the while loop...
   LeftSonar = Double.Parse(telnetClient.read());
   //...
}
```

The Model



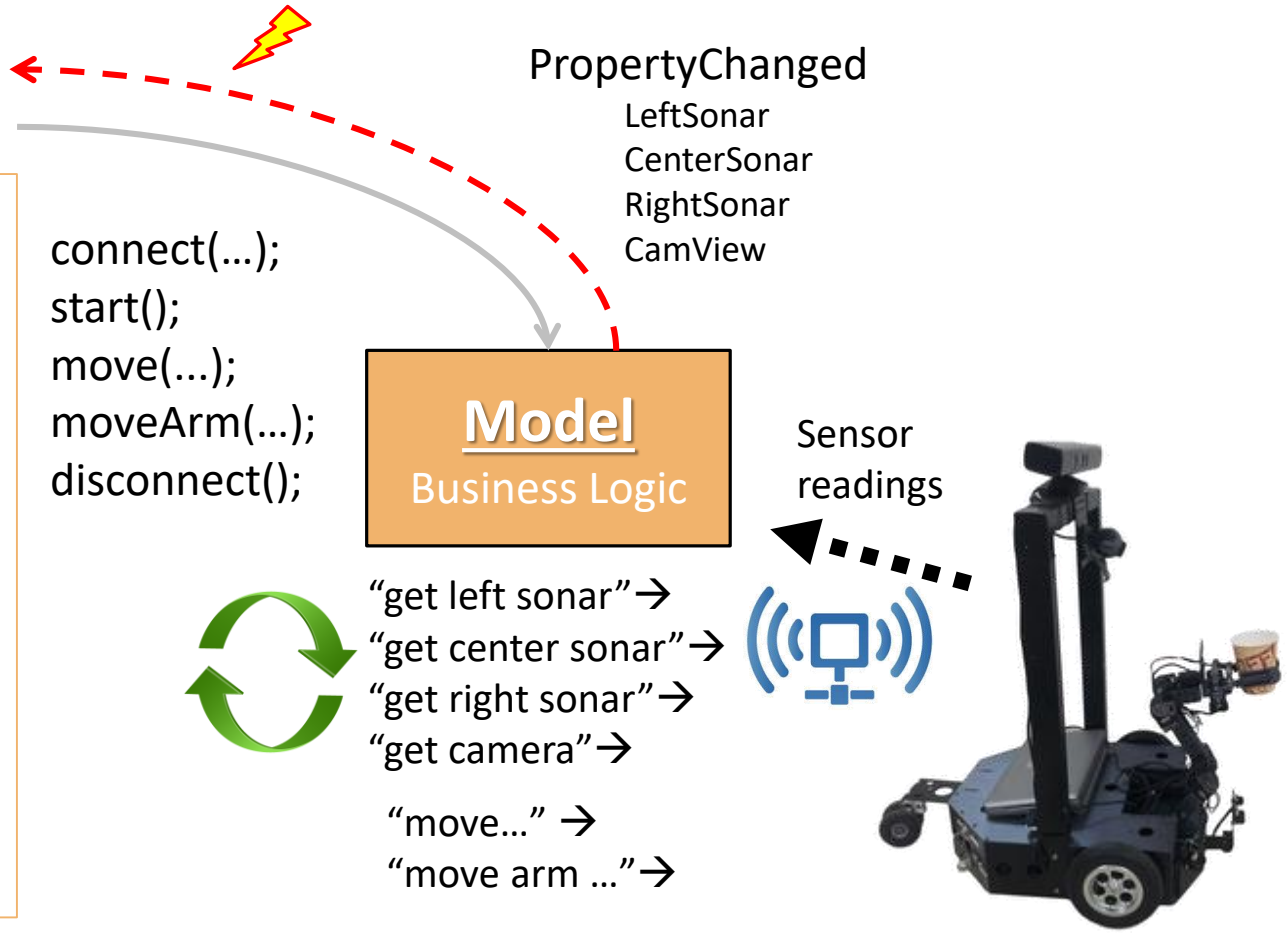
- sending an event that a property has changed

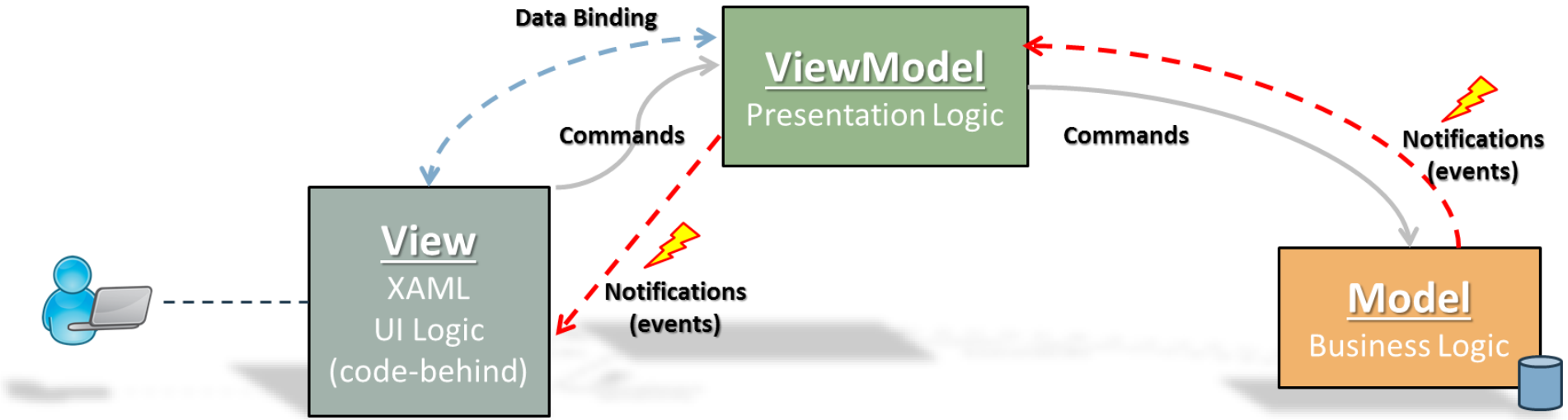
```
// inside MyRobotModel
public event PropertyChangedEventHandler PropertyChanged;

private double leftSonar;
public double LeftSonar
{
     get { return leftSonar; }
     set {
          leftSonar = value;
          NotifyPropertyChanged("LeftSonar");
     }
}

public void NotifyPropertyChanged(string propName){
     if (this.PropertyChanged != null)
          this.PropertyChanged(this, new PropertyChangedEventArgs(propName));
}
```

```
public void start(){
   //... In the while loop...
   LeftSonar = Double.Parse(telnetClient.read());
   //...
}
```

The Model Summery

```
interface IRobotModel  {
     // connection to the robot
     void connect(string ip, int port);
     void disconnect();
     void start();

     // sensors properties
     double LeftSonar { set; get; }
     double CenterSonar { set; get; }
     double RightSonar { set; get; }
     byte[][] CamView { set; get; }

     // activate actuators
     void move(double speed, int angle);
     void moveArm(int az, int e1, int e2, bool grip);
}
```



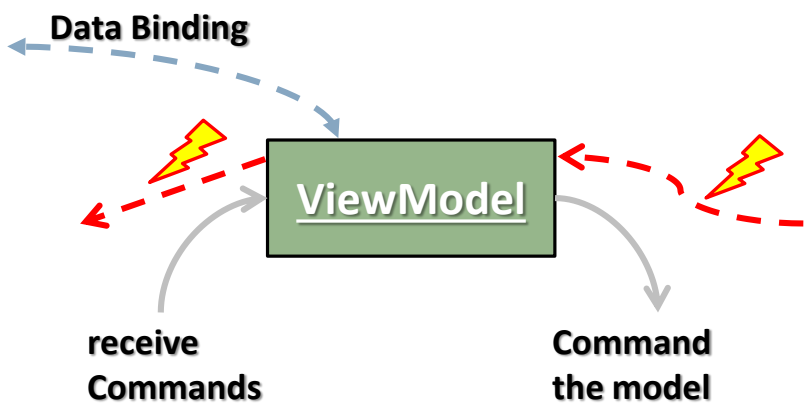


The ViewModel

The ViewModel

- To command the Model we need a reference to a Model
  - We can get notifications from the model by adding delegates to its ***PropertyChanged*** event
- For Data Binding or notifying we need to implement the ***INotifyPropertyChanged*** interface
  - And expose public properties

```
class RobotViewModel : INotifyPropertyChanged {

    private IRobotModel model;
    public RobotViewModel(IRobotModel model) {
        this.model = model;
        //model.PropertyChanged+=...
    }

    public event PropertyChangedEventHandler PropertyChanged;
    public void NotifyPropertyChanged(string propName){...}
}
```

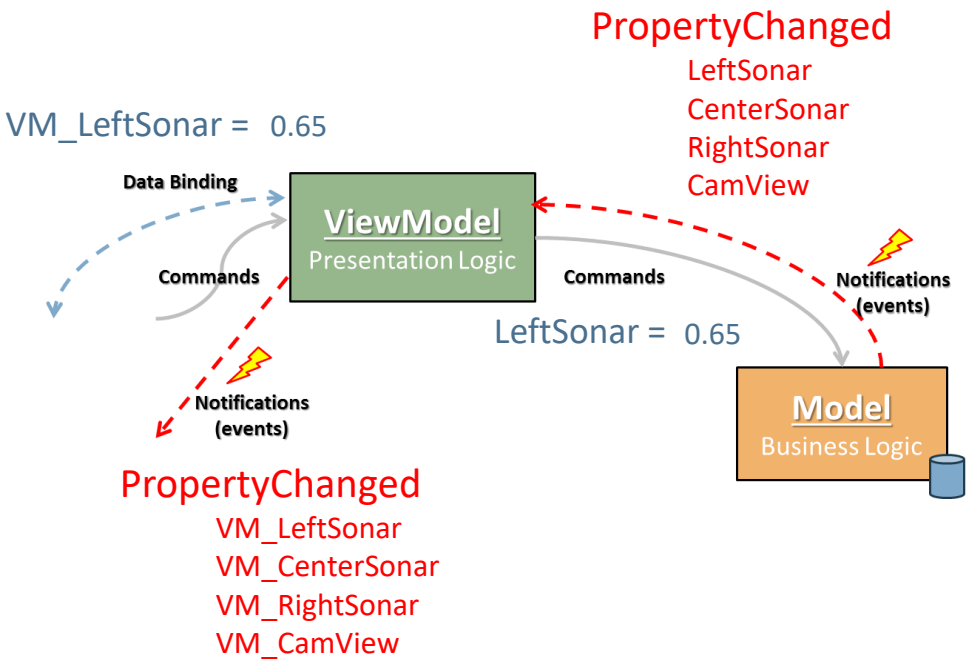


The ViewModel – passing notifications to the view

```
class RobotViewModel : INotifyPropertyChanged {

    private IRobotModel model;
    public RobotViewModel(IRobotModel model) {
        this.model = model;
        model.PropertyChanged +=
           delegate(Object sender, PropertyChangedEventArgs e){
               NotifyPropertyChanged("VM_"+e.PropertyName);
           };
    }
    public event PropertyChangedEventHandler PropertyChanged;
    public void NotifyPropertyChanged(string propName){...}

    // Properties
    public double VM_LeftSonar {
        get { return model.LeftSonar; }
    } // the same for VM_rightSonar, VM_CamView, etc.
```

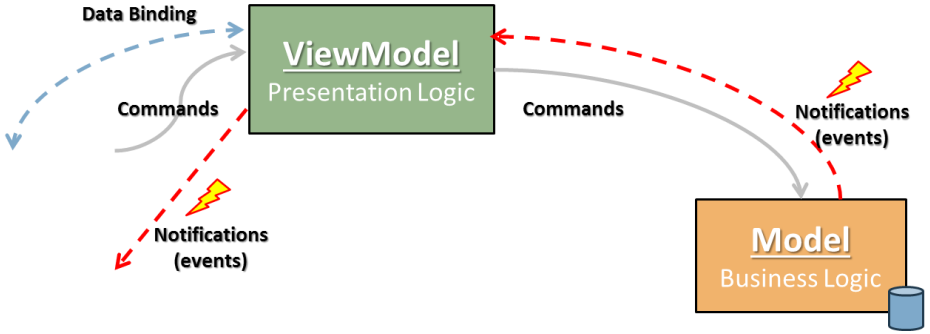
The ViewModel – passing notifications to the view

```
class RobotViewModel : INotifyPropertyChanged {

    private IRobotModel model;
    public RobotViewModel(IRobotModel model) {
        this.model = model;
        model.PropertyChanged +=
           delegate(Object sender, PropertyChangedEventArgs e){
               NotifyPropertyChanged("VM_"+e.PropertyName);
           };
    }
    public event PropertyChangedEventHandler PropertyChanged;
    public void NotifyPropertyChanged(string propName){...}

    // Properties
    public double VM_LeftSonar {
        get { return model.LeftSonar; }
    } // the same for VM_rightSonar, VM_CamView, etc.
```



The ViewModel – Sending Commands

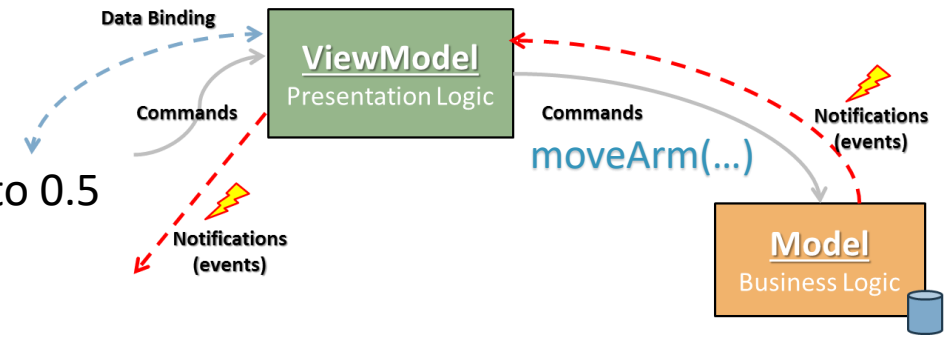
```
// inside RobotModelView
private int angle;
private double robotSpeed;
public double VM_RobotSpeed{
     get { return robotSpeed; }
     set {
          robotSpeed = value;
          model.move(robotSpeed, angle);
     }
}  // the same for VM_RobotAngle

private int az, elv1, elv2;
private bool grip;
public int VM_Azimuth{
     get { return az; }
     set {
          az = value;
          model.moveArm(az, elv1, elv2, grip);
     }
}// the same for VM_Elevation1, VM_Grip, etc.
```



The ViewModel – Sending Commands

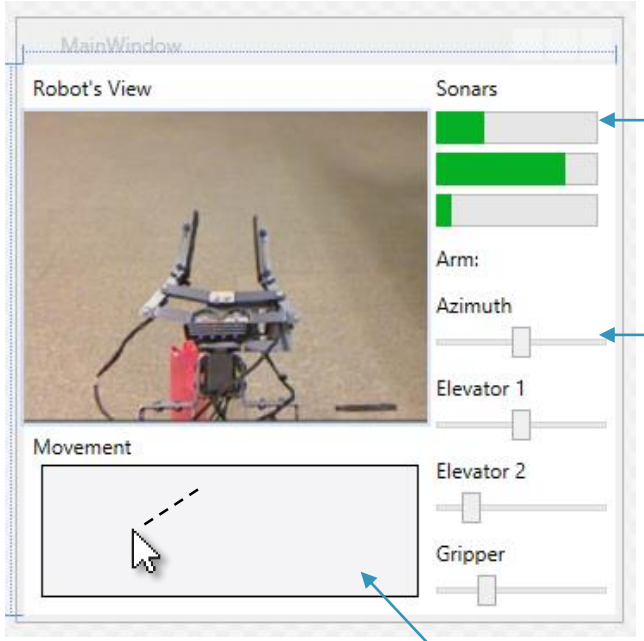
```
// inside RobotModelView
private int angle;
private double robotSpeed;
public double VM_RobotSpeed{
     get { return robotSpeed; }
     set {
          robotSpeed = value;
          model.move(robotSpeed, angle);
     }
}  // the same for VM_RobotAngle

private int az, elv1, elv2;
private bool grip;
public int VM_Azimuth{
     get { return az; }
     set {
          az = value;
          model.moveArm(az, elv1, elv2, grip);
     }
}// the same for VM_Elevation1, VM_Grip, etc.
```

set VM_Azimuth to 0.5




```
public void moveRobot(double speed, int angle){
     model.move(speed, angle);
}
```

The View



data binding with the left sonar

data binding with the arm's azimuth

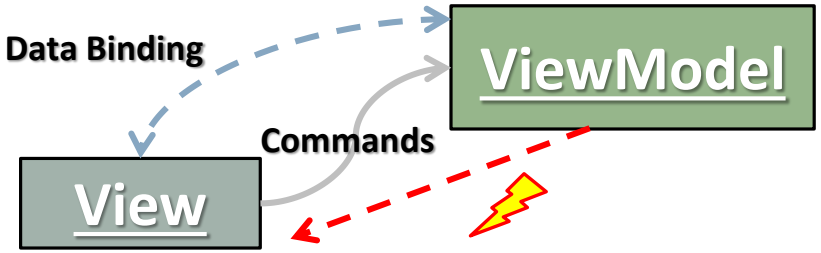
Moving the robot

DATA BINDING EXAMPLE

The View - code behind

- We must know the ***ViewModel*** to command it or to bind with its properties
- The ***Data Context*** of our window should be the ViewModel!
  - This way we can data-bind to its public properties

```
public partial class MainWindow : Window {
    RobotViewModel vm;
    public MainWindow() {
        InitializeComponent();
        vm = new RobotViewModel(new MyRobotModel(new MyTelnetClient()));
        DataContext = vm;
    }
```

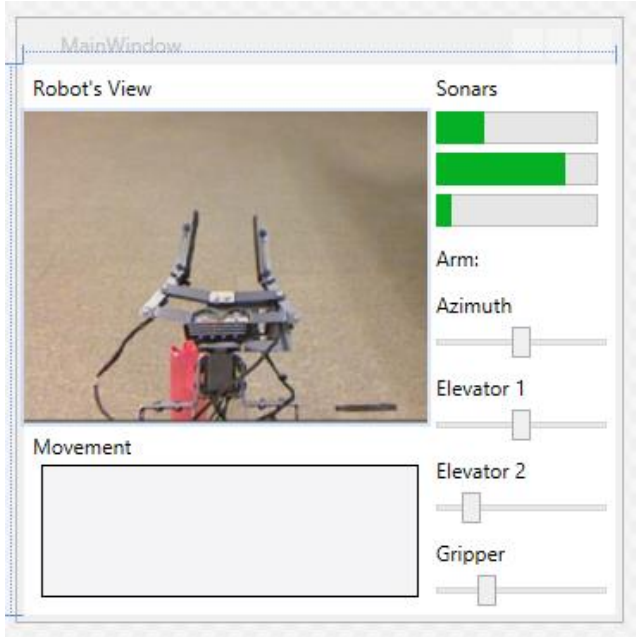


The View - XAML

- In the XAML we simply data-bind to the VM public properties

```
<ProgressBar Value="{Binding VM_LeftSonar}"...
<ProgressBar Value="{Binding VM_CenterSonar}"...
<ProgressBar Value="{Binding VM_RightSonar}"...
```

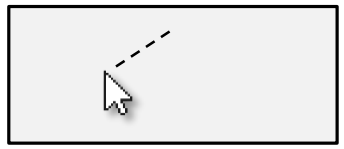
```
<Slider Maximum="60" Minimum="-60" TickFrequency="1"
Value="{Binding VM_Azimuth}"...

<Slider Value="{Binding VM_Elevation1}"...
<Slider Value="{Binding VM_Elevation2}"...
<Slider Value="{Binding VM_Grip}"...
```

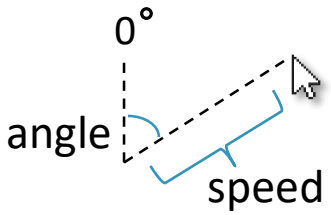


The View – Moving the Robot

- we need to add the following 3 event handlers:

```
<Rectangle
    MouseDown="Rectangle_MouseDown"
    MouseMove="Rectangle_MouseMove"
    MouseUp="Rectangle_MouseUp" .../>
```






```
// in the code behind
private void Rectangle_MouseDown(object sender, MouseButtonEventArgs e){}

private void Rectangle_MouseMove(object sender, MouseEventArgs e){}

private void Rectangle_MouseUp(object sender, MouseButtonEventArgs e){}
```

Calculating the Speed

- The length of the imaginary line should determine the speed
- The maximum allowed speed is 10m/s

```
// data members
double x, y, x1, y1;
bool mousePressed;

private void Rectangle_MouseDown(object sender, MouseButtonEventArgs e) {
    if (!mousePressed) {
        x = e.GetPosition(sender as Rectangle).X;
        y = e.GetPosition(sender as Rectangle).Y;
        mousePressed = true;
    }
}

// helping method
private double length(double x, double y, double x1, double y1){
    return Math.Sqrt((x1 - x) * (x1 - x) + (y1 - y) * (y1 - y));
}
```

Calculating the Speed

```
// data members
double x, y, x1, y1;
bool mousePressed;
```

```
private void Rectangle_MouseUp(object sender, MouseButtonEventArgs e){
    mousePressed = false;
    vm.moveRobot(0, 0);// stop
}

private void Rectangle_MouseMove(object sender, MouseEventArgs e) {
    if (mousePressed) {
        Rectangle r = (sender as Rectangle);
        x1 = e.GetPosition(r).X;
        y1 = e.GetPosition(r).Y; // line: (x,y)---(x1,y1)
        double lineLength = length(x,y,x1,y1);
        double longestDiagnoal = length(0, 0, r.Width, r.Height);
        // normalized speed to be at most 10m/s
        double speed = 10 * lineLength / longestDiagnoal;
        int angle = 0; // calculate at home :)
        vm.moveRobot(speed, angle);
    }
}
```

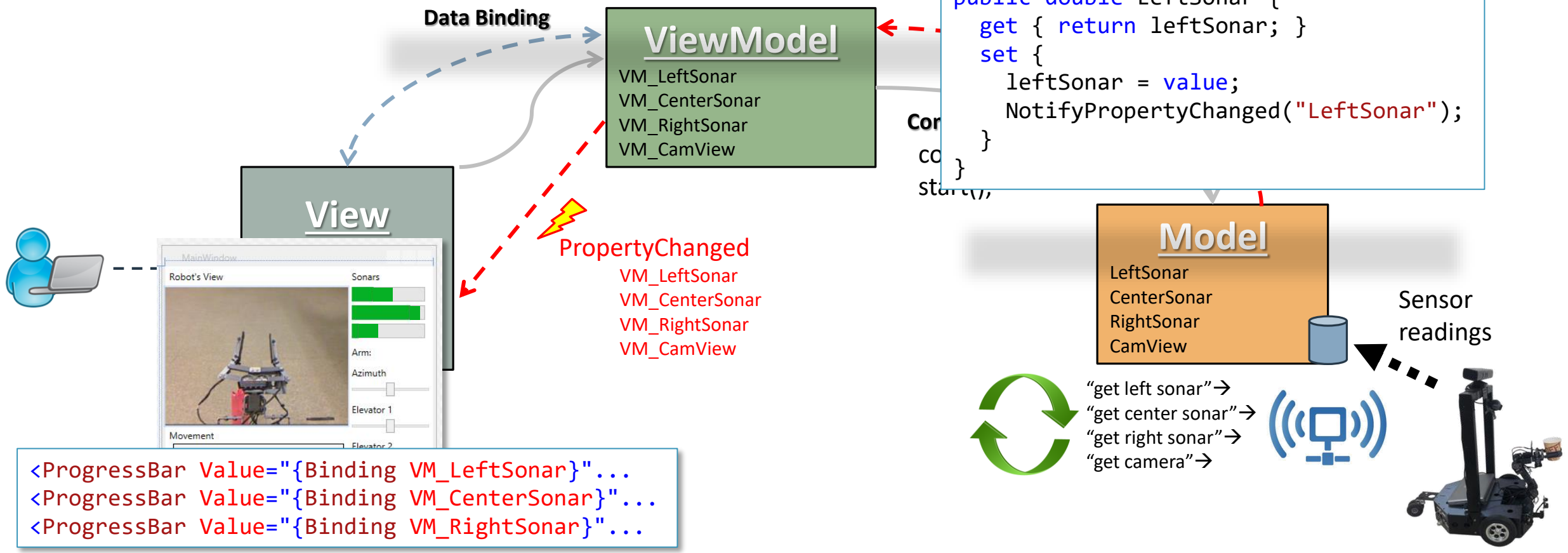
Summary

OUR MVVM IMPLEMENTATION

The MVVM Architecture

```
public event PropertyChangedEventHandler PropertyChanged;
...
model.PropertyChanged +=
     delegate(Object sender, PropertyChangedEventArgs e){
          NotifyPropertyChanged("VM_"+e.PropertyName);
     };
```

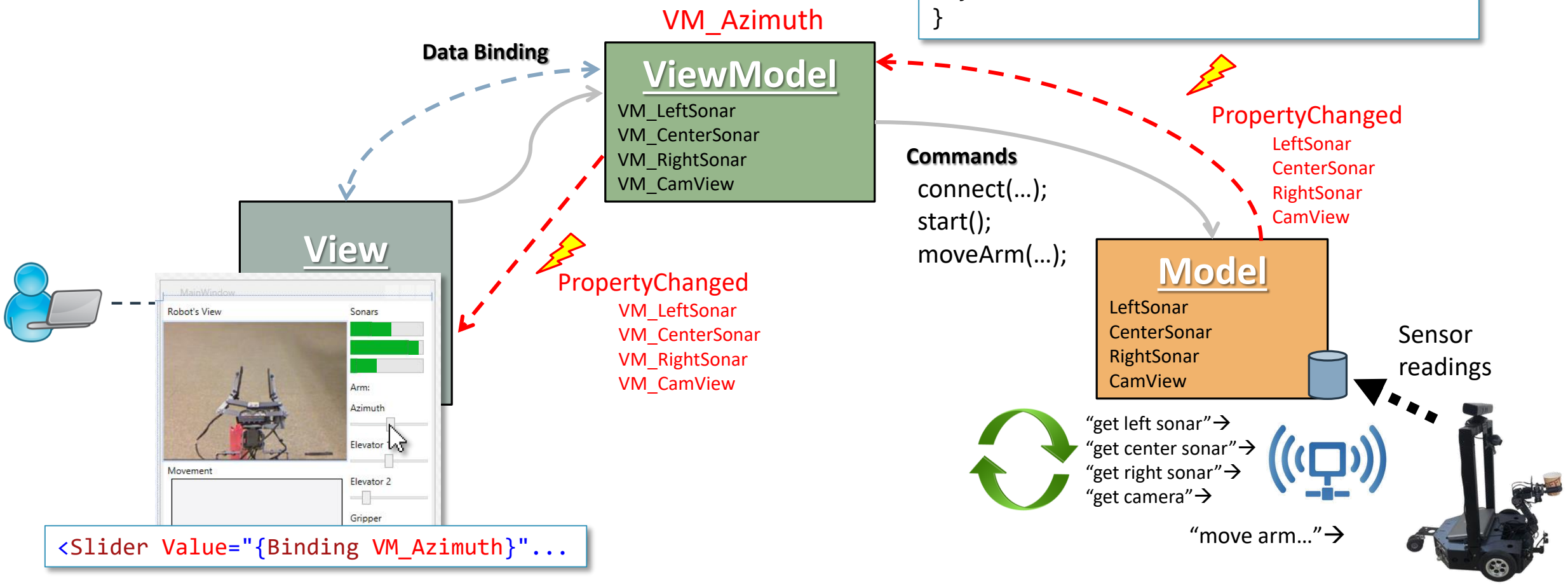
```
public double LeftSonar {
  get { return leftSonar; }
  set {
    leftSonar = value;
    NotifyPropertyChanged("LeftSonar");
  }
}
```




```
<ProgressBar Value="{Binding VM_LeftSonar}"...
<ProgressBar Value="{Binding VM_CenterSonar}"...
<ProgressBar Value="{Binding VM_RightSonar}"...
```

The MVVM Architecture

```
public int VM_Azimuth{
  get { return az; }
  set {
    az = value;
    model.moveArm(az, elv1, elv2, grip);
  }
}
```




```
<Slider Value="{Binding VM_Azimuth}"...
```

The MVVM Architecture