Design Principles for Event-Driven Applications

Cpt S 321

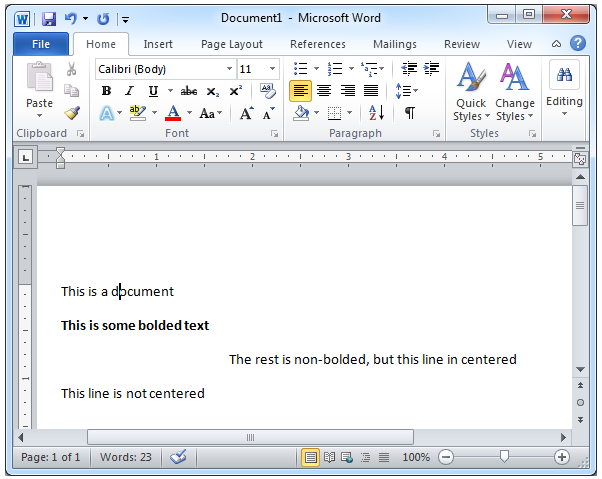
Washington State University

Recall: Event-Driven Applications

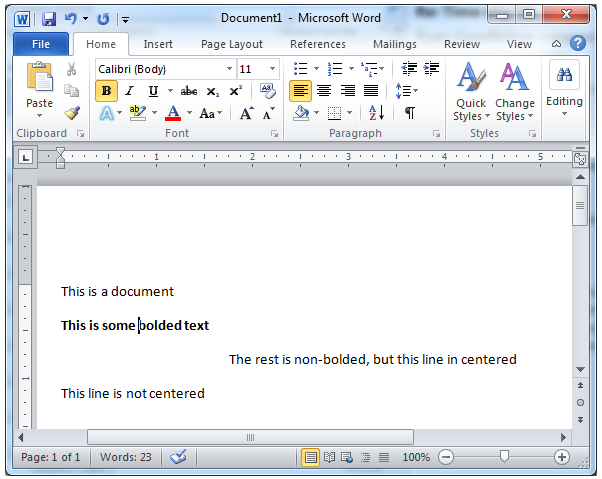
- Recall that event driven applications sit idle until something (an event) happens
- The code you implement in your event-driven applications is just responding to such events
- Consider: You respond to an event such as a button click and make a modification to your application's state and data
  - Application UI must refresh in multiple places to respond to such a change
  - Different components in the application need to know about different state changes and each responds in its own way

Simple Example: Microsoft Word

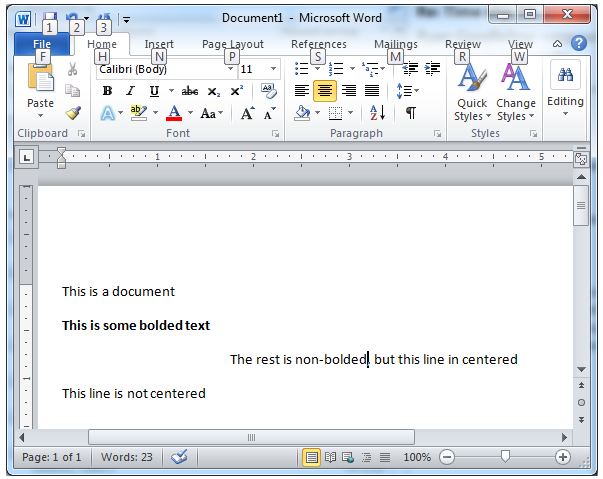
- When you click a location in a Microsoft Word document the cursor moves to the appropriate place in the text. But the toolbar also updates as needed.



Bold button non-highlighted
Left align button highlighted



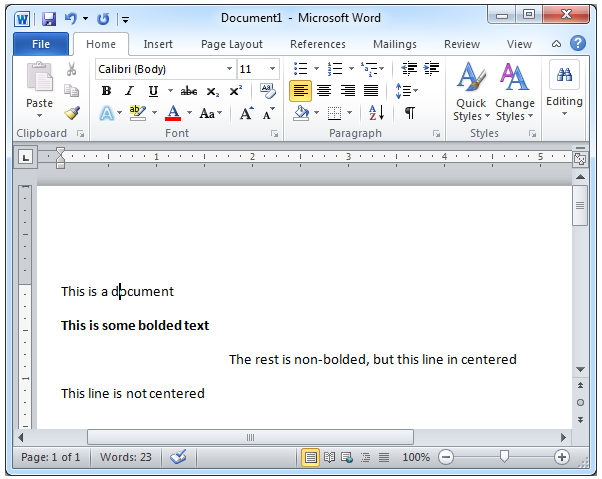
Bold button highlighted
Left align button highlighted



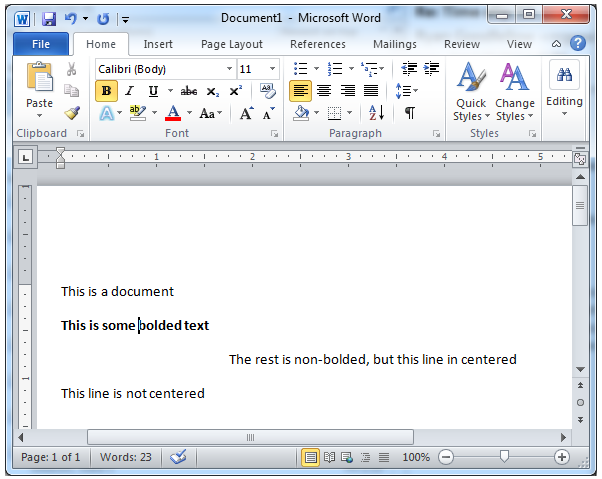
Bold button non-highlighted
Center align button highlighted

Simple Example: Microsoft Word

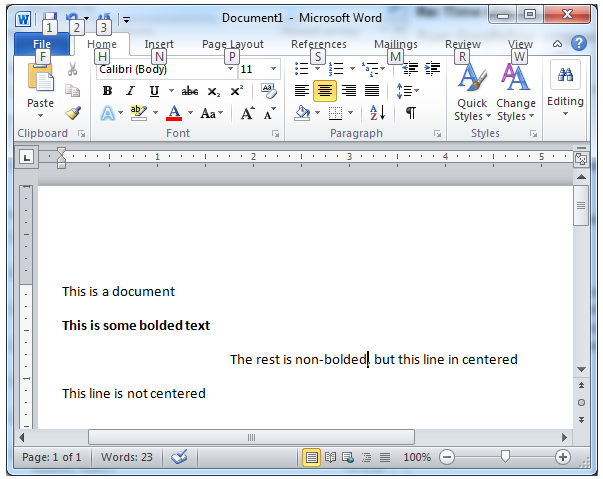
- These same contextual changes in the toolbar would happen if you used the keyboard to change the cursor location.



Bold button non-highlighted
Left align button highlighted



Bold button highlighted
Left align button highlighted



Bold button non-highlighted
Center align button highlighted

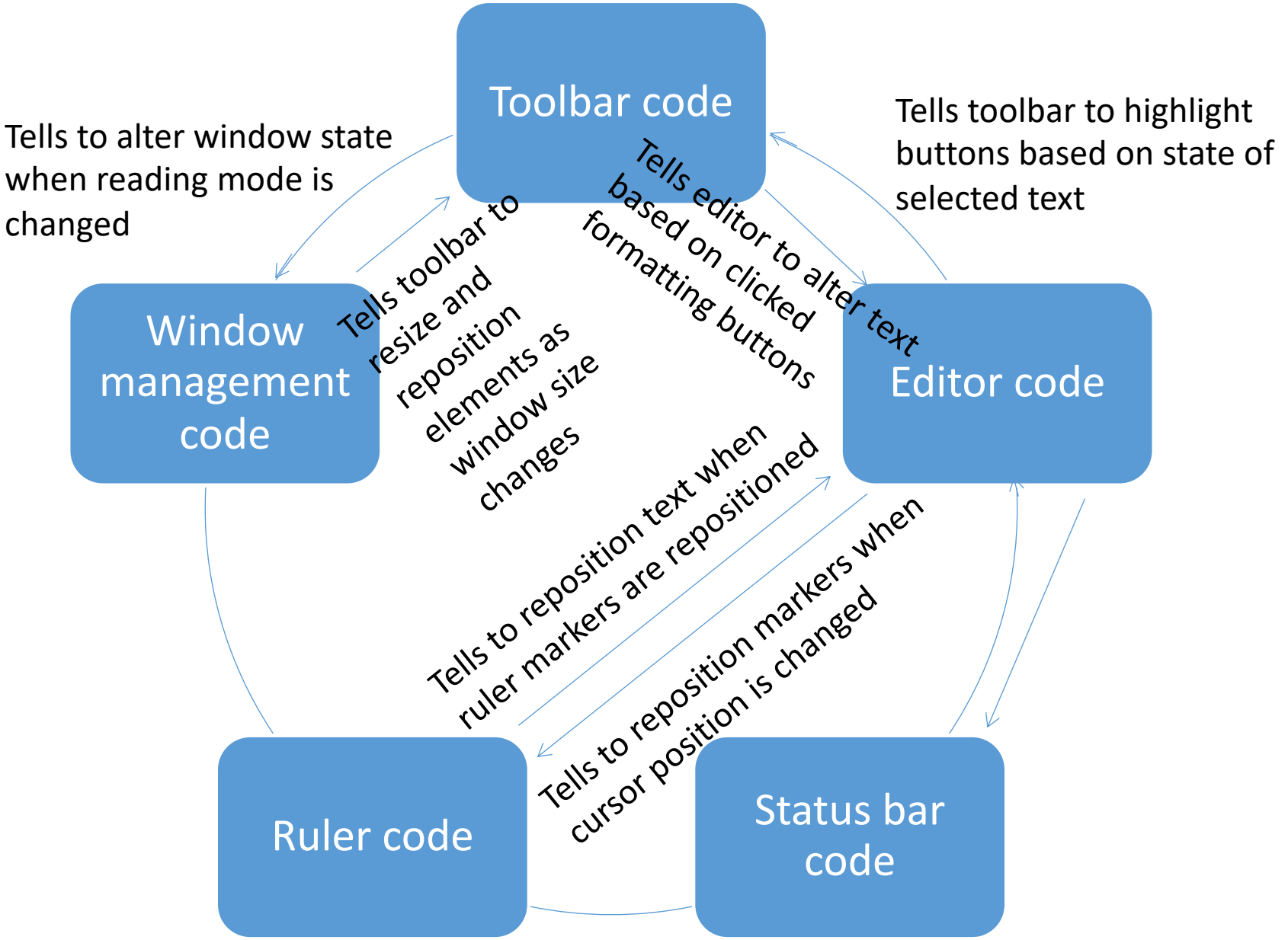
What is NOT happening…

- Think about the code behind this. What is **NOT** happening is this:
  - If (keyboard_changed_cursor)
    - If cursor on bolded text then highlight bold button
    - If cursor on italic text then highlight italic button
    - If cursor on centered text then highlight center-align button
    - …
  - If (mouse_changed_cursor)
    - If cursor on bolded text then highlight bold button
    - If cursor on italic text then highlight italic button
    - If cursor on centered text then highlight center-align button
    - …

Simple Example: Microsoft Word

- LOGICALLY obviously those types of updates mentioned on the previous slide need to happen
- But the design of the application is such that the toolbar (ribbon) code should be decoupled from the editing code
- Consider the diagrams on the following slides

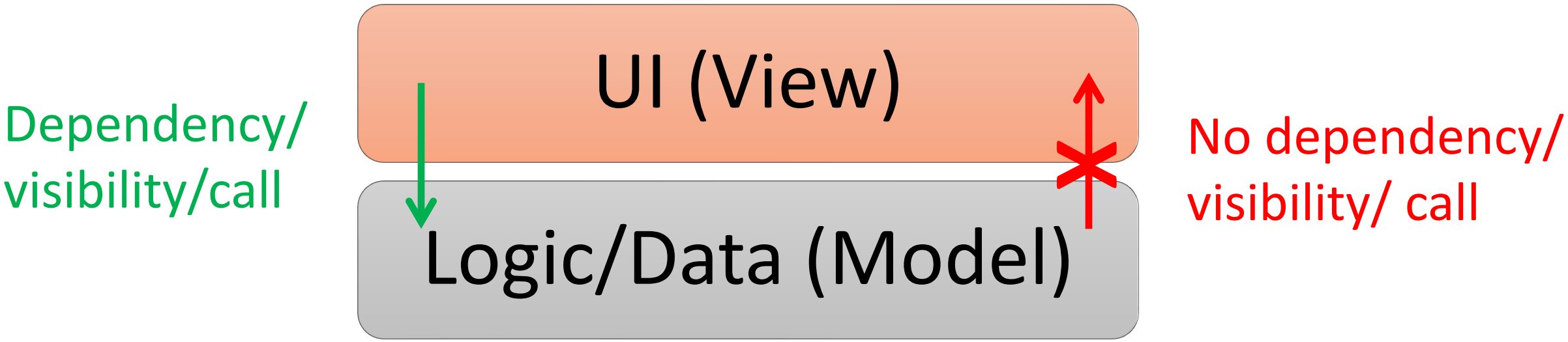
How Too Many Developers Design Applications



Design Complexity

- The previous slide show a VERY small subset of the types of communications between components
- If applications were designed as the previous slide indicates, complexity would go up exponentially as you add more and more features
- This is not how we will design applications in this class
- Start by considering the simple design idea on the next slide

Decouple Logic/Data and UI



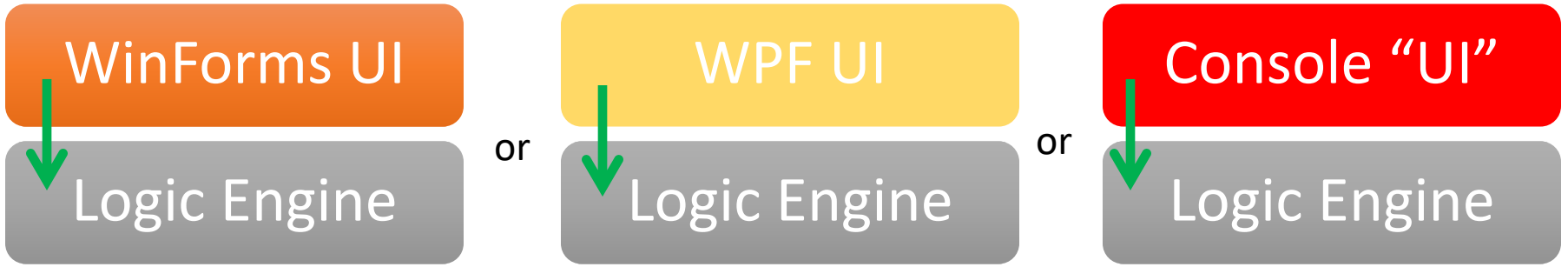
Consider a design that has different key pieces of the application in distinct separable layers

Logic/Data layer

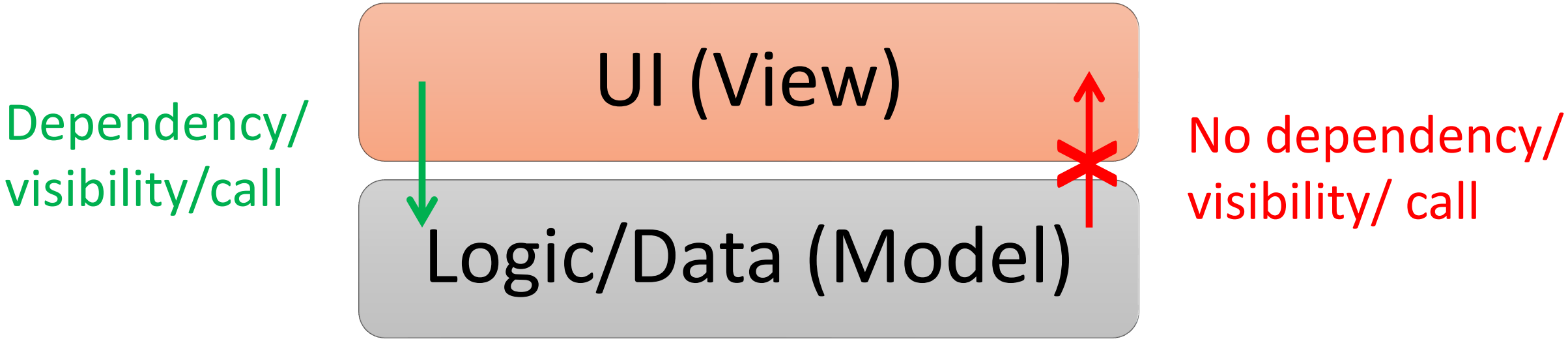
- This layer contains all the logic to perform relevant actions (minus UI interaction) with the data your application is designed to work with
- Take a spreadsheet application as an example. The logic/data layer is the spreadsheet engine that can
  - Load and save spreadsheets
  - Compute formulas
  - Access and update any and all cells in memory
  - Provide information about changes to the spreadsheet(s)
    - We'll see how the C# **event** keyword will assist with this design

UI layer

- Communicates with the logic/data layer
  - Tells it when the user has made alterations
  - Responds to alterations in the underlying layer and updates the interface appropriately
- A key point with the flexibility we're aiming for here is that the UI layer could be swapped out with a new UI layer without needing to make changes in the logic layer



Decouple Logic/Data and UI – WinForms



UI (View): Any user interface code such as Form1 (both the autogenerated one and the one in which you interact with the user).

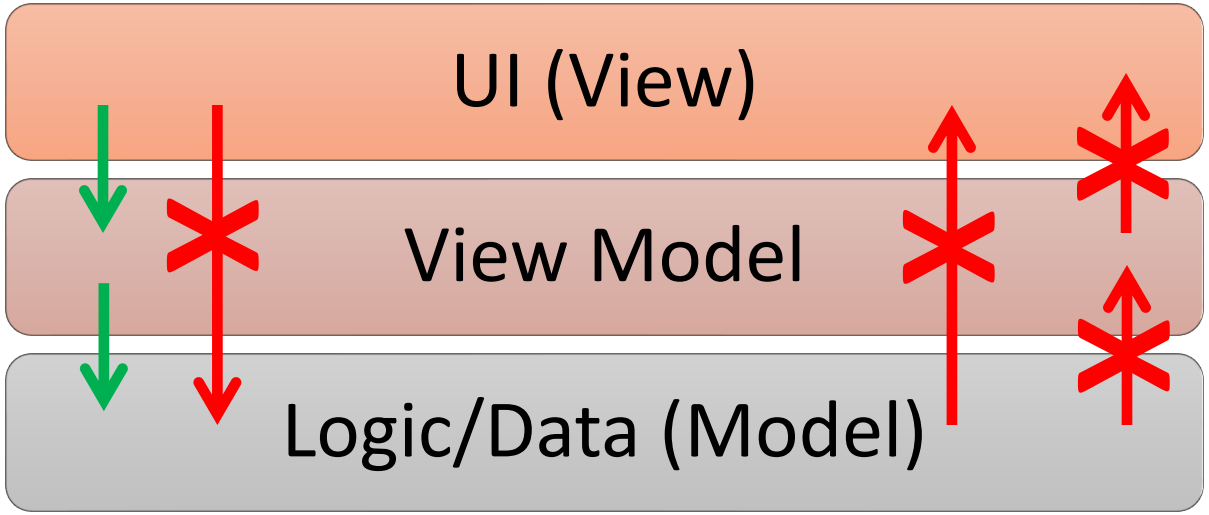
- Ex. from HW2: In the UI layer we had the Form1.cs class.

Logic/Data (Model): Any class that is part of the actual logic of the application.

- Ex. from HW2: In the logic layer we had RandomIntegersListStatistics.cs – i.e., the class that implemented the 3 ways to determine distinct integers.

Decouple Logic/Data and UI – Avalonia through the Model-View-ViewModel (MVVM) architecture

Dependency/ visibility/call



No dependency/ visibility/ call

UI (View): User interface code such as MainWindow.axaml in which we interact with the user.

- Ex. from HW2: MainWindow.axaml class in which we defined the TextBox.

View Model: An abstraction of the model (like a wrapper) that exposes only what is needed from the model.

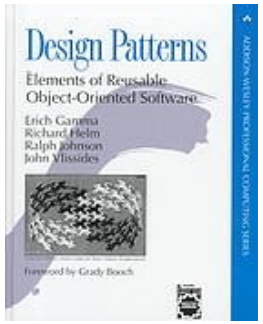
- Ex. from HW2: MainWindowViewModel.cs in which we only exposed a method that build a string with the statistics.

Logic/Data (Model): Any class that is part of the actual logic of the application.

- Ex. from HW2: RandomIntegersListStatistics.cs (same as the previous slide).

Software design principles and patterns

- Design principles: general guidelines on how to design applications with a “better” design
- Design patterns: “In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design.” Wikipedia
- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 978-0-201-63361-0.



Objects that Notify of Property Changes

- There is a **design pattern** (the observer design pattern) where you write a class (or struct) such that it notifies other sources when its state changes
  - Roles:
    - Broadcaster: The class that is changing (ex. a BankAccount in the context of a banking application)
    - Observers/Listeners/Subscribers: The types that are notified when the broadcaster has changed (ex. different parts of the interface that need to be updated with the change)
  - Sometimes it provides notifications for changes on just a few of its properties
  - Other times it can notify if ANY aspect of its state has changed

Objects that Notify of Property Changes

- Look at this simple example:

```
public class MyClass
{
    public string name;
}
…
MyClass mc = new MyClass();
mc.name = “Some new name”;
```

- How do we implement MyClass so that it will be able to “notify others” when the `name` field changes?

- Is a field what we want here? What’s the other option for the name member declaration if we want the usage to remain the same?

Objects that Notify of Property Changes

- Name should be a property, not a field. That way we can add behavior, i.e., implement code to do something when the property value gets changed

- What does a “notification” really mean and how many things might want to be notified when the Name property changes?

```
public class MyClass
{
    public string Name
    {
        get; set;
    }
}
MyClass mc = new MyClass();
mc.Name = “Some new name”;
```

Objects that Notify of Property Changes

```
public class MyClass
{
  private string name = “(not named)”;
  public string Name
  {
    get { return name }
    set
    {
      // What goes here?
    }
  }
}
```

```
public class MyClass
{
  private string name = “(not named)”;
  public string Name
  {
    get { return name }
    set
    {
      if (value == name) { return; } // Important. Why?
      name = value;
      NotifyAllListenersOfPropertyChange(“Name”);
      // Placeholder method above, but how to implement?
    }
  }
}
```

Observer/Listener/Subscriber

- Before we finish the class implementation, let us define what we want: a list of observers/listeners/subscribers that will be notified when the property changes.

- This can just be a list of function pointers/references. Then **outside sources can add functions to this list and those functions will be called when the property changes**.

- This design is simple and can be done in a variety of languages. **Objects that support notification of property changes have a list of function pointers/references internally and call all functions in that list when a property is changed.**

We COULD do it like this…

DO NOT TRY TO COMPILE

```
public class MyClass
{
  private List<FuncPtr> listeners = new List<FuncPtr>();
  private string name = “(not named)”;

  public void AddListener(FuncPtr funcRef) { listeners.Add(funcRef); }

  public string Name
  {
    get { return name }
    set
    {
      if (value == name) { return; }

      name = value;
      foreach (FuncPtr ptr in listeners) { ptr.Invoke(“Name”); }
    }
  }
}
```

A made-up thing

… but there's a better way to do this in C#

- C# has two things that allow us to deal with this situation
- First off, note that I just made up "FuncPtr" in the previous example, but there's something in C# called a **delegate** that serves as a function pointer
  - Represents a pointer (reference) to a function that has a specific signature (return type and parameter list)
- Second, we don't need to manually keep a list of function pointers because C# has the **event** keyword, which internally is really just

  a list of delegates

Delegates

- In most scenarios you'll be able to use a delegate that is already defined in the .NET framework
  - Can define your own if you need to, but we'll focus on using existing ones in this class
- PropertyChangedEventHandler Delegate on MSDN
- Exists for this exact type of design scenario
- Remember the concept: this is a delegate declaration, meaning that when you have an instance of the **`PropertyChangedEventHandler`** at runtime, it is a reference to a function with a specific signature

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

Events

- C# events tutorial on MSDN

- Internally this is a list of delegates
  - add to this list with the **+=** operator (i.e., **subscribe** to the event)
  - remove from the list with **-=** (i.e., **unsubscribe** to the event).
  - No further list functionality exists
  - Designed to be protected
  - The functions in that list can only be called by the class that contains the event

Finishing MyClass (broadcaster)

```
public class MyClass : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged = delegate { };
  private string name = “(not named)”;

  public string Name
  {
    get { return name }
    set
    {
      if (value == name) { return; }
      name = value;
      // The event gets called as if it were one function (in terms of syntax)
      // Calls everything (all subscribed delegates) in the internal list
      PropertyChanged(this, new PropertyChangedEventArgs(“Name”));
    }
  }
}
```

INotifyPropertyChanged Interface

- Since designing objects to notify of property changes is such a common thing, an interface already exists in C#: INotifyPropertyChanged Interface
- Implement this interface if you want a class or struct to be able to notify outside sources (subscribers) of property changes
  - READ THE MSDN PAGES
  - There's a lot of information there on how to implement this interface and how to use events. Will need to know this well to implement your spreadsheet application.

Updating an observer when MyClass has changed

```
public partial class Form1: Form | class MainWindowViewModel: ViewModelBase
{
        private MyClass instanceOfMyClass;

        public Form1() | MainWindowViewModel()
        {
                InitializeComponent();
                // Initialize a person object
                instanceOfMyClass = new MyClass("");
                // Subscribe to the property changed event
                 instanceOfMyClass.PropertyChanged += MyClass_PropertyChanged;
                // Change the name to fire the event
                instanceOfMyClass.Name = “MyNewName”;
        }
         private void MyClass_PropertyChanged(object sender,
                                              PropertyChangedEventArgs e){
             // Write the code here to update whatever needs to be updated.
        }
}
```

Avalonia: The ViewModel adding an extra layer

- Remember that the ViewModel is an abstraction of the Model and as such it manipulates the model and notifies the UI about changes, meaning that the **ViewModel need to propagate any changes in the Model**
- An advantage of this is that the ViewModel is then **testable**!
- The easiest way to do that is to use ReactiveUI by having the ViewModel inherit from ReactiveObject (by default it does – check the ViewModelBase class that the ViewModel inherits from)

```
public class MyViewModel : ViewModelBase
{
        private string caption;
        public string Caption
        {
                get => caption;
                set => this.RaiseAndSetIfChanged(ref caption, value);
        }
}
```

Avalonia: Bindings using ReactiveUI

In the XAML code we can bind controls to, for example:

- Properties

  <Button Content="{Binding NameButtonText}" …
  where NameButtonText is a property declared in the ViewModel

  - To trigger the update of the UI in the ModelView if we are certain that the property has changed:

    this.RaisePropertyChanged(nameof(NameButtonText));

  - To trigger the update of the UI only if a field has changed:

    this.RaiseAndSetIfChanged

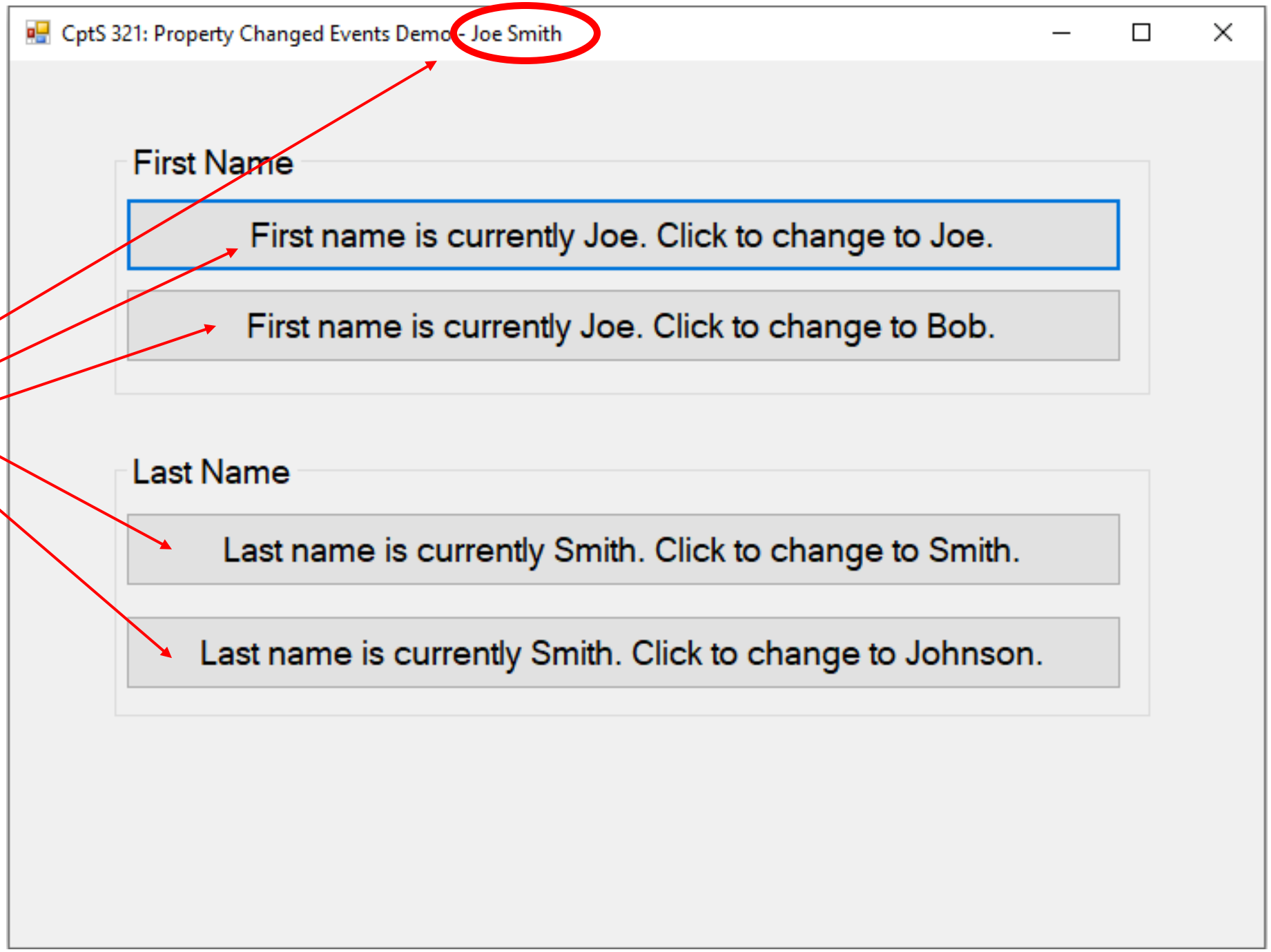
- Commands

  < Button Command="{Binding ButtonNameOnClickCommand}"
  CommandParameter="myNewName" …

  Will call the method implemented in the ModelView when the button is clicked:
  private void ButtonNameOnClickCommand(string aNewName){…}

Code demo (cont.)

- Initial value;
- Updated with button clicks



What do we need to do to make it work?

- Link Buttons to the specific names:
  - WinForms: Using the UI Designer, set the tags of the 4 buttons to Joe, Bob, Smith, and Johnson, respectively. (Set the tags in the Properties)
  - Avalonia: Use command parameters in your .axaml code

    <Button Command="{Binding ButtonFirstNameOnClickCommand}" CommandParameter="Joe" …
- In the class **`Person`** (both WinForms and Avalonia)
  - Implement **`INotifyPropertyChanged`**
  - By implementing the interface a **`PropertyChangedEventHandler`** event will be auto-generated
  - Implement the set properties in a way that when the property value changes, we notify all listeners

What do we need to do in the View ?

- In the class **`Form1`** (WinForms) / .axaml (Avalonia)
  - Define two fields (as usual private):
    - **`person`** (an instance that will be updated) and
    - **`windowTitlePrefix`** (a string that will not change)
  - In the constructor of the View for WinForms (Form1) / ViewModel in Avalonia (MainWindowViewModel):
    1. Create a new person with no name (i.e., empty strings)
    2. Subscribe to the property change event by adding the following method to the event: **`Person_PropertyChanged`** (event handler; pay attention on the signature – check slide 23)
    3. Set the person's name to Joe Smith using the Properties

```
person.FirstName = "Joe";
person.LastName = "Smith";
```

What do we need to do in the View ? (cont.)

- In the the View for WinForms (Form1), implement the following methods:
  - **`btnFirstName_Click`** (updates the person's first name), signature:

    ```
    private void btnFirstName_Click(object sender, EventArgs e)
    ```
  - **`btnLastName_Click`** (updates the person's last name)
  - **`Person_PropertyChanged`** (sets the buttons' and the form's text)

- In the the ViewModel in Avalonia (MainWindowViewModel), implement:
  - **`ButtonFirstNameOnClickCommand`**(updates the person's first name):

    ```
    private void ButtonFirstNameOnClickCommand(string newFirstName)
    ```
  - **`ButtonLastNameOnClickCommand`** (updates the person's last name)
  - **`Person_PropertyChanged`** (sets the buttons' and the form's text)

    ```
    private void Person_PropertyChanged(object? sender, PropertyChangedEventArgs e)
    ```