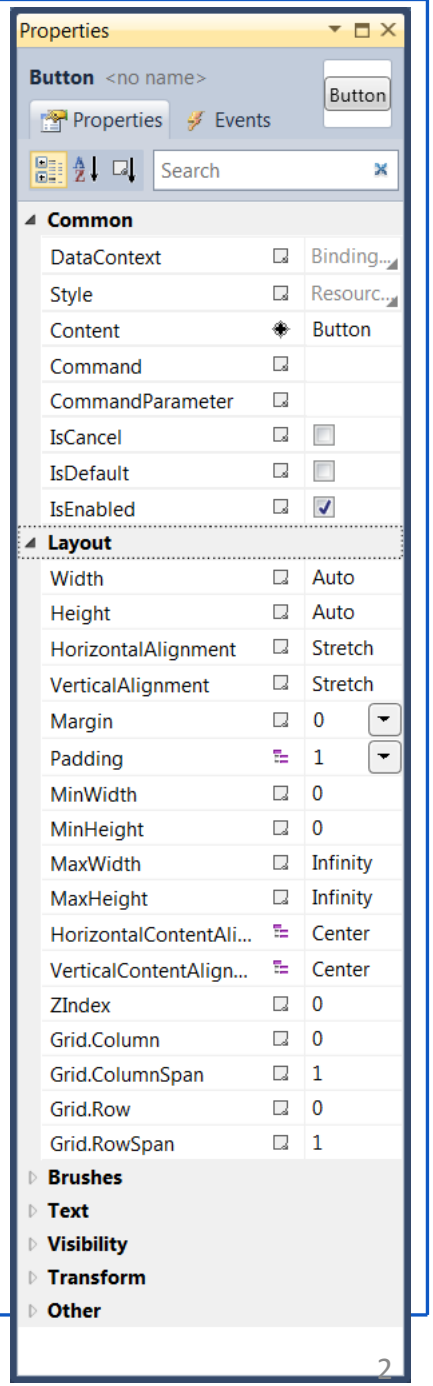
Dependency Properties

.NET Properties

- You know ordinary .NET property implementation uses a backing member variable, like this:

```
public class SomeControl {
  private Color _background;
  public Color Background {
    get {return _background;}
    set {
      _background = value;
      Invalidate(); // Force repaint
    }
  }
}
```

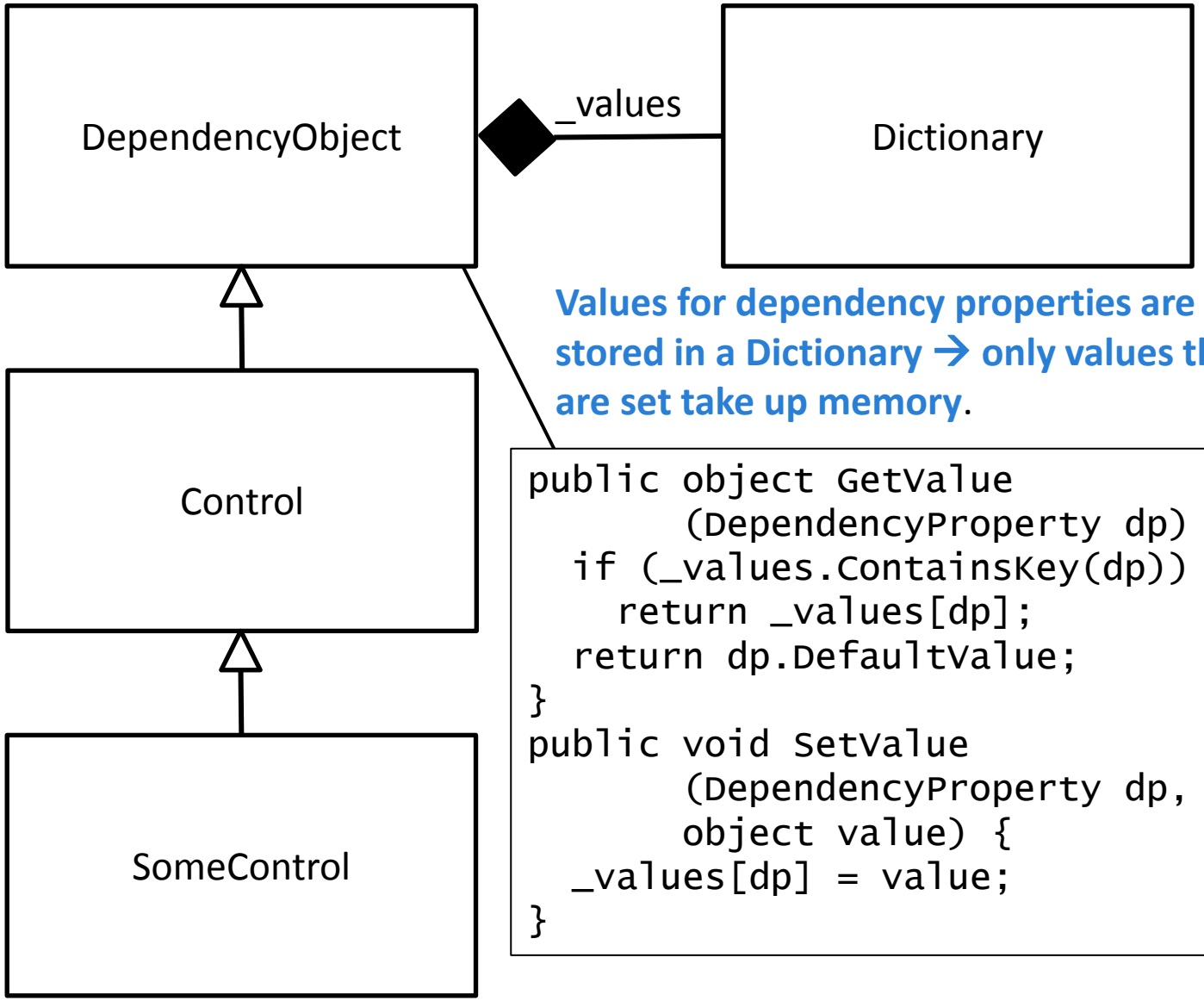
- And this is a superb implementation in many cases.
- But if you consider that a control (widget) in WPF has 40 – 70 properties and often only a couple is used to store a value
  → this is a waste of memory
  - and due to the GC that moves objects around, this will also affect performance!



WPF Property System - Goal

- The WPF team decided to enhance the ordinary property implementation.
- Their goal was:
  1. **Sparse storage**
  2. **Styling**
  3. **Animation**
  4. **Binding**
  5. **Attached properties**

WPF Property System - Architecture



Values for dependency properties are stored in a Dictionary → only values that are set take up memory.

```
public object GetValue
       (DependencyProperty dp) {
  if (_values.ContainsKey(dp))
    return _values[dp];
  return dp.DefaultValue;
}
public void SetValue
       (DependencyProperty dp,
       object value) {
  _values[dp] = value;
}
```

DependencyObject

```
public class DependencyObject : DispatcherObject {
  IDictionary<DependencyProperty, object> _values =
              new Dictionary<DependencyProperty, object>();
public DependencyObject();
public ClearValue(DependencyProperty dp) {
  _values.RemoveValue(dp);
  }
public object GetValue (DependencyProperty dp) {
  if (_values.ContainsKey(dp))
    return _values[dp];
  return dp.DefaultValue;
}
public void SetValue (DependencyProperty dp, object value) {
  _values[dp] = value;
protected virtual void OnpropertyChanged (PropertyChangedEventArgs
e) {}

// Many methods and implementation details omitted for the
// audiences sanity
}
```

Defining a Dependency Property

- You can add dependency properties only to classes that derive from DependencyObject.
  - Most of the key classes of WPF infrastructure does that.
- The first step is to define an object that *represents* your property. This is an instance of the DependencyProperty class.
- The information about your property needs to be available all the time, so your DependencyProperty object must be defined as a static field in the associated class.
- E.g.:

```
public class FrameworkElement: UIElement, ...
{
public static readonly DependencyProperty MarginProperty;
...
}
```

Registering a Dependency Property

- You need to register your dependency property with WPF.
- This step needs to be completed before any code uses the property, so it must be performed in a static constructor for the associated class.
- But first, you must create a FrameworkPropertyMetadata object that indicates what services you want to use with your dependency property

```
static FrameworkElement()
{
FrameworkPropertyMetadata metadata = new FrameworkPropertyMetadata(
            new Thickness(),
            FrameworkPropertyMetadataOptions.AffectsMeasure);

MarginProperty = DependencyProperty.Register("Margin",
            typeof(Thickness), typeof(FrameworkElement), metadata,
         new ValidateValueCallback(FrameworkElement.IsMarginValid));
...
}
```

Adding a Property Wrapper

- The final step to create a dependency property is to wrap it in a traditional .NET property.

```
public class FrameworkElement: UIElement, ...
{
  ...
  public Thickness Margin
  {
    set { SetValue(MarginProperty, value); }
    get { return (Thickness)GetValue(MarginProperty); }
  }
}
```