XAML Guidelines

Introduction

One of the most frequently asked questions from WPF practitioners is “What are the recommended practices for organizing a WPF project?” When I hear this question, I shrug my shoulders and answer “It depends; every project is different”. I still feel that is the accurate answer, but a slightly more useful answer for those new to the platform is to share some of the common practices, and let them select the ones that fit their project. This write-up covers my personal experiences and the experiences of many people I have worked with in the past. The good suggestions are theirs, and the bad suggestions are mine.

The goals of this paper are:

- To share an empirical reference for those just starting.
- To encourage others to share their practices and challenge the suggestions herein. These guidelines are not comprehensive—let us know what I missed.

Why XAML guidelines?

A little organization, consistency and planning around your XAML provides many benefits, among them:

- Easier code maintenance, improved reuse, and improved collaboration among team members.
- Better performance at run time.
- Minimize overhead for the tools and optimize the designer experience, which increases productivity and decreases time to market.

Come on, get to the guidelines. Please!

Here is a link list to navigate the topics, the hierarchy is not perfect, I have to recommend you read it in order.

Contents

Introduction 1

Why XAML guidelines? 1

Come on, get to the guidelines. Please! 1

Overall Project Organization 2

Naming conventions 3

Scenario 1: Naming elements within a scene 3

Scenario 2: Naming elements within a template 4

Organizing resources 5

Scenario 1: Organizing logical resources within a scene 5
Scenario 2: Organizing resources in a resource dictionary 6
Using static versus dynamic resources (in non-themed projects) 8
Using static versus dynamic resources (in themed or skinned projects) 8
Using the xmlns prefix 8
Using implicit styles versus explicit styles 9
Wrapping control templates in styles 9
Templating inside Expression Blend 10
Referencing file-based resources (dictionaries, images, fonts, etc.) 10
Declaring individual XAML elements 10
Converters 11
Commands 11
Theming (or more commonly skinning) 12
Conclusion 13
Thanks 13

Overall Project Organization

I will not touch much on project refactoring and packaging (creating an .exe versus a library). Regardless of the type of binary you are creating, you must decide how to organize the many different types of assets that WPF applications use. The obvious practice here is to create subdirectories to keep items organized. A typical structure follows:

- **\Resources** is the directory for most of your resources. This directory should include only XAML files[1] and subdirectories.
- **\Resources\Images** or **\Images** should contain the icons and other images that are included as resources in the assembly. These items should be included as type "Resource," not "Embedded resource".
- **\Resources\Fonts** or **\Fonts**, include the fonts embedded as resources in the assembly.
- **\Converters** is for value converters. This directory includes only class files. One converter per file is preferred.

[1] Technically, ResourceDictionaries can have code-behind class files, but that practice is discouraged.

- **\Commands** is the directory for your commands - classes that implement **ICommand** or custom routed commands-. This directory should include only class files.
- \T**hemes** includes generic.xaml, other theme-specific resource dictionaries, and theme specific subdirectories. I use the following guidelines:
    - When working on projects that are not themed, I leave the default structure generated by the tools – this means including default styles for custom controls in generic.xaml.
    - If working on a themed project, include a resource dictionary for each theme. Use the <themename>.<theme color>.xaml convention (e.g. luna.normalcolor.xaml)
    - When working on a skinned project, follow same convention than themes and include a resource dictionary for each skin in the \Themes directory.
    - If a theme or skin is not self-contained and needs other resource dictionaries or other resources, create a subdirectory in \Themes and put the extra resources files there. Each subdirectory is named after the theme or skin.
        - These "theme-specific"subdirectories can have subdirectories such as \Fonts and \Images.
    - Note that any resources shared among multiple themes still goes under the non-theme specific directories (e.g. \Resources\Fonts or \Fonts).
- **\Other** is my catch-all to avoid too many extra directories. I usually group other user interface logic or helpers into this directory. For example, I use the other subdirectory to contain freezables that carry **DataContext**; or I use it for template selectors, or dependency objects that override existing UI elements' metadata.

Naming conventions

There are three aspects to naming elements: what to name, how to name it, and the actual naming convention. There are also two scenarios for naming: elements within a scene and elements within a template, in particular a **ControlTemplate**.

Scenario 1: Naming elements within a scene

What elements to name

For scenes that will be designed by a human designer (not auto-generated and not just by a developer), all logical elements in the scene should be named. Even if the elements will not be referenced in code, if they are in the view and represent a logical UI object, they probably deserve a name.
Be aware this does have a performance implication: the compiler generates a member in the partial class for your scene and it assigns the instance of the XAML element to this member, it registers the name in the scope, and it increases the size of your assembly size a tiny bit.

Notice the emphasis on "logical" when I referred to named elements. For the most part, this means any Control, UserControl, Panels, or any other Framework Element that will used in data binding or animation. The type of element that "logical" does not include is art work. For example, if you have an icon built of three paths, you will name only the logical entity, and not name the three paths. This logical entity would likely be the container for the artwork. That said, artwork should be in brushes as much as possible.

How to name the element

When assigning a name, use the **x:Name** attribute for consistency. Using the **Name** attribute works for **FrameworkElement** and **FrameworkContentElement**, and many people could argue[2] that this covers almost everything you will use in WPF. However, I like **x:Name** better, because it is easier to spot in relation to other XML attributes

What naming convention to use

The naming convention is still up for debate. Here are my preferences:

- Be consistent in your naming.
- Use Pascal casing. It is more readable, it involves less typing of unnecessary prefixes, and almost truly represents the behavior, because the generated members are internal.
- Provide meaningful names. Instead of a name like `border1`, use something like `TrackBackgroundBorder`.
- Make the **x:Name** attribute the first attribute after a type instance, like this:

```
<Button x:Name="CancelButton" ... />
```

- Postfix names with the type of the element. For example, use a convention like `x:Name="CustomerListBox"` or `x:Name="SearchTextBox"`.

Scenario 2: Naming elements within a template

Within a template, name only the elements that absolutely need names. This includes the template parts that the control expects, the elements used in triggers, animations, named bindings, and so on. This is usually enough to keep the XAML readable.The justifications for the differences between template and scene naming practices are:

- Control templates are smaller than scenes, so there is less to keep track of and XAML should be easy enough to navigate
- Use names in templates emphasize critical parts that are used for storyboards, bindings, or template parts; names make it easier for anyone overriding the template to know what is critical and what other elements that can be removed.

Another difference between this scenario and naming elements in the scenes is the use of the `PART_` prefix for template parts that are expected by the control. All the parts that the control expects and cannot work without should have the `PART_` prefix. Nothing else should have the prefix.

The rest of the advice for this scenario is the same than naming elements in scenes: use the `x:Name` attribute—put that attribute first, use Pascal casing, and so on.

---

[2] I'm not sure if the Visual Studio WPF designer team would argue it, but the designer defaults to **Name**, so I am explaining it on their behalf.

Organizing resources

Resource organization also varies based on scenarios: the resources within a scene, the resources in a resource dictionary, and the resources in generic.xaml.

Before getting into resource organization, I need to discuss the different philosophies of separating the resources into files. I group these into two approaches:

- A physical or type-based organization where all instances of a type are grouped together. For example, a resource dictionary would include all constants (or metrics), a separate file for all colors, another file for brushes, and so on. I tried this a few times, and it did not work very well for me. It leads to redundancies where you end up including all files and duplicating resources all around. It is also harder to navigate across references (for example, from the color to the brush).
- A logical organization where resources are grouped on files to accomplish a logical task. For example, a group includes all resources for a theme, or all resources used for a specific set of controls (such as charts) are grouped together as a logical entity. The resources files will include constants, colors, brushes, styles, etc. This approach has proven more efficient on many projects therefore it is the one discussed in this writing.

Scenario 1: Organizing logical resources within a scene

The resources within a scene are those local to a window, user control, or page. They are usually included at the root element of the scene (for example, in a **Window.Resources** element). I use the following guidelines for organizing these resources:

- If there are dependencies between resources, the resources should be consistently ordered. Reusable resources should be at the top of the resource dictionary. A common structure usually follows the order shown in the following example:

```
<Window.Resources>
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <!-- #Constants -->
    <!-- #Colors    -->
    <!-- #Brushes   -->
    <!-- #Converters -->
    <!-- #Objects (such as Data) or commands (for ribbon),etc. -->
    <!-- #Styles     -->
    <!-- #DataTemplates  -->
</Window.Resources>
```

- Each section for each type should be labeled with a comment that indicates the type of the resources it includes. I also like to ‘prefix’ each section name with `#` or `##`, this helps you quickly find these section headers by using CTRL+F.

- Resources that are not shared should be declared in line within the scope of the element that contains them. This raises the question of why you would use something that is not shared as a resource. Reasons include the following:
    - To pass context to elements that are declared as properties of an element and are not in the tree. For example, you might use Freezable to pass DataContext from a **DataGrid** control to its **Column** objects.
    - To declare a variable that you might reuse, such as a brush.
- If a resource is shared among just a few elements, it can be moved closer to the elements that use it. Try not to do this too often. It can be handy, but if you overuse this approach, you will end up with no structure.

How to name the resources

- All resources should have an **x:Key** attribute. Silverlight allows you to use the **Name** attribute instead of **Key**, but I advise that you avoid that practice for consistency with WPF.
- For resource keys, use Pascal casing and follow the same meaningful postfix convention we used for x:Names. I do not use the type as a postfix for metrics, for geometries, or for drawings. For example, I would not use the key `LabelWidthDouble`, I would use `LabelWidth`. Similarly, I would not use `CompanyLogoGeometry`, but instead would just use `CompanyLogo`.
- When creating implicit styles, use the **x:Key** attribute, as shown in the following example:

    ```
    <Style x:Key="{x:Type Button}" TargetType="{x:Type Button}">
    ```

    This reads consistently, and makes it easy to search for `x:Key="{x:` if you want to go find implicit styles. The WPF compiler does allow you to skip it, but try not to skip it.

Scenario 2: Organizing resources in a resource dictionary

Resource dictionaries are organized in the same order as the resources in a scene and use the same naming conventions. I will not touch on that again. However, I will discuss tasks that are specific to **ResourceDictionary** objects.

Keep resources files small (but not too small)

Generic.xaml and other theme resource dictionaries can get large. If you don't want to manage large files, you can partition them into smaller resource dictionaries and use the **MergedResourceDictionary** feature to pull them all together. For App.xaml, using **MergedResourceDictionaries** is a must. It really helps navigate the logical organization.

Organizing resources in generic.xaml

For generic.xaml, resource organization depends on the project. If the project is small enough, I like to keep generic.xaml as a standalone file.

That said, few projects are small enough, so when using merged resource dictionaries that will go into generic.xaml, I include just **MergedDictionaries** in generic.xaml, and then add .generic to the name of the dictionaries , this emphasizes the difference between the resource dictionaries that will go into generic.xaml and the ones that will go into scenes.

For example, the name Charts.generic.xaml means the resource dictionary contains the styles and

control templates for charts, and these will be merged into generic.xaml and should not be included in App.xaml or the scene.

Merging resource dictionaries (the XAML order)

- When merging resource dictionaries, follow the order of reuse, with the most-reusable items at the top.
- When merging a resource dictionary and local (scene) resources, always include your local resources last, as shown in the following example:

```
<ResourceDictionary>
  <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="shared.xaml" />
  </ResourceDictionary.MergedDictionaries>
  <SolidColorBrush x:Key="duplicate" Color="Pink" />
</ResourceDictionary>
```

  In this case, if `duplicate` is a key that is already in shared.xaml, you will still get the local copy, which is probably what you want. Duplicate resources are not a best practice and you should detect them at design-time, but in case you miss it, following this order can help. To test and catch errors, switch the order or use tracing.

Merging resource dictionaries (for optimal sharing)

Merging resource dictionaries by using XAML is easy to do in Microsoft Expression Blend. Unfortunately, the way Blend does it does not always result in the behavior you want at run time. The problem is when you use syntax like the following to merge in XAML:

```
<Window.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="pack://application:,,,/Dictionary1.xaml" />
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Window.Resources>
```

This creates too many separate instances of your resource dictionaries. In this case, for example, two instances of `Window1` will each have their own private copy of **ResourceDictionary**. If there were a `Window2` instance with a reference to shared.xaml, both `Window1` and `Window2` would instantiate this resource dictionary and use their own private copy. To get around this duplication, do the following:

- In your application, merge your resource dictionaries in App.xaml.
- For projects where you do not have access to App.xaml (for example, a control library) or for scenarios where a scene includes resource dictionaries that are not in XAML, merge resource dictionaries by using code. Alternatively, create a subclass for **ResourceDictionary** that merges resources dictionaries by picking resource dictionaries from a singleton that caches the **ResourceDictionary** instance.[TODO: blog example on sharedRDs] .

[Editorial note, this closes the section on organizing resources. Is it clear from structure or should we add conclusion]

Using static versus dynamic resources (in non-themed projects)

Everyone's recommendation is to use **StaticResource** references whenever possible because of performance. I do follow the practice, but have to admit that several attempts to detect the performance overhead have not been successful. Another (stretch) benefit for using **StaticResource** elements is the ability to catch bugs at development time—because **StaticResource** objects throw exceptions when they can't resolve, this is a good way to detect mistakes such as typos.

That said, I am not overzealous about just using **StaticResource** elements for the following reasons:

- Expression Blend generates only **DynamicResource** and I don't like to keep fighting the tool, and I usually miss some.
- Seldom, Blend cannot resolve a **StaticResource** reference even though it exists and is in the right place. The symptom is that the application behaves fine at run time, but Blend is not resolving the resource. Because of this, I might leave a few **DynamicResource** references in the application when it helps me avoid an error in Blend so I can design the screen. However, this happens very rarely.

Using static versus dynamic resources (in themed or skinned projects)

If you are skinning or theming, you will need some **DynamicResource** elements. This does not mean that all resources must be dynamic. There are plenty of resources that can still be referenced as **StaticResource** elements, such as resources that are local to a template, local to a window, and so on.

If you are theming, you also need to make sure that any resources in the **Styles** elements in the themed dictionary (like generic.xaml) are referring either to **StaticResource** elements or to **ComponentResourceKey**. **DynamicResource** elements will not resolve in a style in a themed dictionary.

I discuss this issue later in this paper under Theming.

Using the xmlns prefix

The **xmlns** prefix maps from a CLR namespace (and assembly reference) to an XML prefix so the XAML processor can identify and instantiate the type. When creating xmlns mappings, follow these guidelines:

- Keep the prefix short but meaningful. Four to five characters is a good (yet flexible) rule. For anything that is not a system or tools prefix, two or more characters is preferred—leave **x:**, **d:**, etc. for the system and tools. Create more meaningful prefixes for your mappings.
- Use meaningful prefixes. This can be the last part of the CLR namespace you are referring to, or a combination of library name and namespace. Common abbreviation techniques are to use the first letters of a namespace, or to remove vowels from the name and sound it out. For example, `MyLibrary.DataGrid` might become `dtgrd`, and `MyLibary.ViewModel` can become `vwmdl`.
- Be consistent. Use the same abbreviation across all XAML files, and use a consistent abbreviation technique.

A common exception for the names is the practice of using **xmlns:local** for references to the local namespace in the project. I avoid that because **local** then means something different for each assembly. I also don't embrace **local** because too often my assemblies have multiple CLR namespaces.

If you are a component vendor, implement the attribute for inserting a namespace automatically (the attributes are assembly:XmlnsDefinition and assembly:XmlnsPrefix). Here you can go over the suggested five-character limit because your abbreviation is likely not local to all customers and languages. Users will be able to override it if they need to keep it short. When adding the namespace attribute, feel free to use lowercase letters; that tends to be the norm for **xmlns** declarations. For example, blendable controls from IdentityMine ship with an **xmlns** value of `blendables`.

Using implicit styles versus explicit styles

An implicit style is a style that is aimed at overriding all instances of a type. The following example shows the XAML for an implicit style:

```
<Style x:Key="{x:Type Button}" TargetType="{x:Type Button}"> ... </Style>
```

If you need a consistent look and feel throughout an application, this feature can save you a lot of needless explicit style references all over your XAML. The inclusion of resource dictionaries that declare implicit styles is most often done in the App.xaml file. This makes sense, because this location represents the broadest application scope. However, there is a bug that shipped in the .NET Framework 3.0 and has been left in the system for backward compatibility. What happens is that implicit styles included in App.xaml will take precedence over scene or element styles when you use a **DataTemplate**. For the most part, for people who use implicit styles and want a consistent look and feel, this is not a problem. But it can be an issue for those who use implicit styles and then try to override them with new local styles. The workaround is to not include the style in App.xaml and include it instead in the scene. It then behaves as expected. Another workaround is to override the style in the **DataTemplate** itself.
TODO: Blog example on the above because it would take a page to explain it here.

Another practice to avoid is to declare implicit styles as resources within **ControlTemplate** of a control that can have children or content inside it. Very occasionally you have to do this, but be aware of the side effects for the content of the control.

Wrapping control templates in styles

A lot of people wonder why Expression Blend always wraps a **Control** template in a style. This is not required by the WPF run-time, so the question becomes whether you should also wrap the **ControlTemplate** in a style. Answer: Yes! The benefits are as follows:

- Consistency. There will be plenty of times where you do need the style in order to pass parameters to a template, so you might as well keep all of the templates consistent by wrapping all of them with a style.
- Sanity check on your templates. If you are creating a control template and do not have any properties that you are setting by using the style (**Foreground**, **Alignment**, etc.), your template

might be too restrictive (in fact, you can consider it hard-coded). A good rule of thumb is to use the style to pass the parameters that you expect the designer to override.

- Less friction. Fighting Blend (or any other designer tool) is not easy, by leaving the Templates wrapped in the Styles, you will avoid a lot of hand tweaking.Inheritance. I like that styles have inheritance functionality (that is, they support **BasedOn**), which can create some nice flexibility for you.

Templating inside Expression Blend

A common pitfall for designers is letting Blend hard-code references to themes. When you select **Edit Template**, Blend defaults to the theme of the operating system you are using at design-time. This means you are starting with a non-generic template and are adding references to OS-specific themes. A common workaround is to use Simple Styles, which is a generic resource dictionary that is included in Blend that is not tied to an OS theme and that has simpler templates.

Unfortunately, the Simple Styles theme is not as rich as other themes, so you might not be able to use it all of the time. When you do use an OS theme, you can still carefully remove references to the theme in specific elements. You can also use a less complex and more generic style like Classic. To have Blend use Classic style for templates, the easiest approach is to change your OS theme to match the theme you want, because there are a few known bugs that prevent you from having custom themes included manually into Blend.

Referencing file-based resources (dictionaries, images, fonts, etc.)

There are several choices for referencing resources: whether to use absolute paths or relative paths, whether to pack syntax or not, and so on.

I like absolute paths. You can move (or cut and paste) XAML from one file to another, which often crosses directories. For example, you might move an object from the UserControls directory to the Resources directory; if you had used relative paths, the reference would be broken.

I also use the **pack://** syntax, despite it being a little cumbersome. This syntax is explicit about the location for the resource. Without pack, it is possible to create a dependency from a DLL to the main application. For example, a DLL might link to \Resources\images\fromapp.png and the code will compile and run just fine. However, if you use the DLL in another application, the dependency will be broken.

In short: when using resource references, be as explicit as you can.

Declaring individual XAML elements

I already covered the basics of naming an element, but what about the rest of the options for defining an element? In particular, should you use multi-line or single-line declaration? This practice is usually optional or not strongly enforced, because if you do use it, you will spend lots of time fighting the tool. For those coding all XAML by hand, or wanting to follow a convention, I have observed the following:

- Designers who have to read the XAML like to split it into multiple lines. They don't like to scroll horizontally. And they also have less horizontal space, because Blend has toolbars on both sides of the design and editing window.
- Developers like to keep as much XAML in a single line as practical, so they don't have to scroll vertically. I also think developers don't tweak the markup as much, so their XAML is less verbose and requires less horizontal scrolling.

Here are some compromise guidelines:

- Do not use a hard rule of one line per **UIElement** or one property per line.
- When splitting the properties into multiple lines, group them into categories such as layout, appearance, content, and so on, as shown in the following example:

```
<Button x:Name="SubmitButton"
  HorizontalAlignment="Right" VerticalAlignment="Bottom" Margin="0,0,30,20"
  Foreground="{StaticResource ..}" BorderBrush="{StaticResource …}" ...
  Command="{Binding SubmitCommand}"
</Button>
```

- When declaring a XAML element that has no content inside it, use a self-closing element (that is, close the tag with **/>**) For example, use `<TextBlock ... />` instead of `<TextBlock ... ></TextBlock>`
- Indent the attributes within an element. I have no strong preference between tabs and spaces. I use spaces most often simply because tabs indent more than I want.
- For properties that are data-bound and have a long binding expression (for example, `Source=`, `Converter`), put the binding in its own line. If the expression is long enough to warrant two lines, I indent the second line with respect to the binding-expression line.

Converters

Converters are usually declared as resources (because they are referenced in XAML). Declare them as early in the Resource dictionary as practical, one per type. Use the type name as the **x:Key** name, such as `<local:BoolToVisibilityConverter x:Key="BoolToVisibilityConverter" />`

A logical approach is to include these in the App.xaml file. I like resource dictionaries to be idempotent, so I include the converters in the resource dictionary if the styles or data templates in the resource dictionary need them. This might cause a little redundancy, but it is very minor.

Commands

Avoid instantiating and sharing commands in XAML. Commands should be either data bound to a **ViewModel** or reference an **x:Static** command handler outside of the view. An exception to this rule is the **Ribbon** control. This control supports the hybrid concept of a **RibbonCommand**, which is a command with lots of view state. I think of **RibbonCommand** more as a view rather than as a command, so I declare it in XAML. The WPF team is changing this approach and **Ribbon** commands will not be defined this way in the next CTP of **Ribbon**.

Theming (or more commonly skinning)

All the tips presented earlier in the resources sections apply to projects whether they are being skinned or not. In this section I will highlight the differences when you are using themed resource dictionaries.

- The \Themes directory is the standard location for your default styles (which by default are in generic.xaml); do keep it there for consistency. Generic.xaml should be your "backup" style. In addition, follow these guidelines:
    - If you have just one resource dictionary per theme, you can include all the dictionaries in the themes directory. The naming convention for the directory is usually the <themename>.<themecolor>.xaml
    - If your themes consist of several resource dictionaries, a subdirectory for each theme is recommended. However, as mentioned below keep the generic.xaml file in the default location. I like to name each subdirectory using the same convention as the resource dictionary (minus the .xaml). For example:
      \Themes\luna.normalcolor\luna.normalcolor.xaml
      \Themes\luna.metallic\luna.metallic.xaml
- Shared resources in a themed resource dictionary should be declared with **ComponentResourceKey**, not with a resource key.
- Avoid declaring **ComponentResourceKey** values in XAML.
  That means that you should not use syntax like this:

```
<SolidColorBrush x:Key="{ComponentResourceKey
TypeInTargetAssembly={x:Type
local:SomeType},ResourceId=OtherArbitraryName}" Color="Green"/>
```

  Instead, declare a static class that contains the keys and use an **x:Static** attribute to refer to these in XAML. In a .cs file, use code like the following:

```
public static class SharingRDKeys
    {
        public static ComponentResourceKey ButtonBrushKey
        {
            get
            {
                return FindOrCreateKey(typeof(SharingRDKeys),
KeysEnum.ButtonBrush);
            }
        }
    }
```

  In the generic.xaml or any other theme file, us markup like the following:

```
<SolidColorBrush Color="Green" x:Key="{x:Static
local:SharingRDKeys.ButtonBrushKey}"/>
```

- You cannot declare implicit styles in a theme. It just does not work.

- Make sure you understand the differences between dynamic and static resources. The best advice here is to open a theme from the Blend\SystemThemes directory and analyze it.

  The following examples show you that this guideline is not all black and white:

  - Theme resource dictionaries are not all dynamic references. Only the resources that are going to change by theme are dynamic.
  - If you are creating a **StaticResource** reference to a resource with a **ComponentResourceKey**, you might not be using it the way the theme designer expected.
- If you are using themed resource dictionaries, you should merge them at the application level by using the App.xaml file or code.

Conclusion

Organizing your XAML, being consistent, and following some conventions throughout your project will make your code more maintainable, easier to navigate, and maybe even more efficient at run time. There are many ways to organize your project; I hope the content in this paper helps you think about and plan your approach.

Thanks

- Special thanks to Jonathan Russ, Nathan Dunlap, and Jared Porter from IdentityMine. These folks participated in a small interview-like session where they shared their practices. Sprinkled throughout the document you find their wisdom.
- Unni Ravindranathan from the Blend team also subjected himself to the interview. Also Unni (and Pete Blois) have indulged my silly questions over the years on dealing with Blend, so huge thanks to them and the rest of Blend team.
- Mike Pope, who got the hold of the document and in no time returned it completely red-lined with useful edit suggestions.
- Paul Stovell wrote the original XAML guidelines. This is a great start and if you have seen the interviews I did for the series, I tried to start from there. I also think Paul inspired customers to ask Microsoft for better guidelines, which indirectly pushed me to get this out.
- All the many people with whom I have worked with in past WPF projects. This includes several agencies and many big companies and ISVs (that will remain nameless to avoid the need to ask for PR approval).
- [TODO: Add any feedback/reviewers people.]