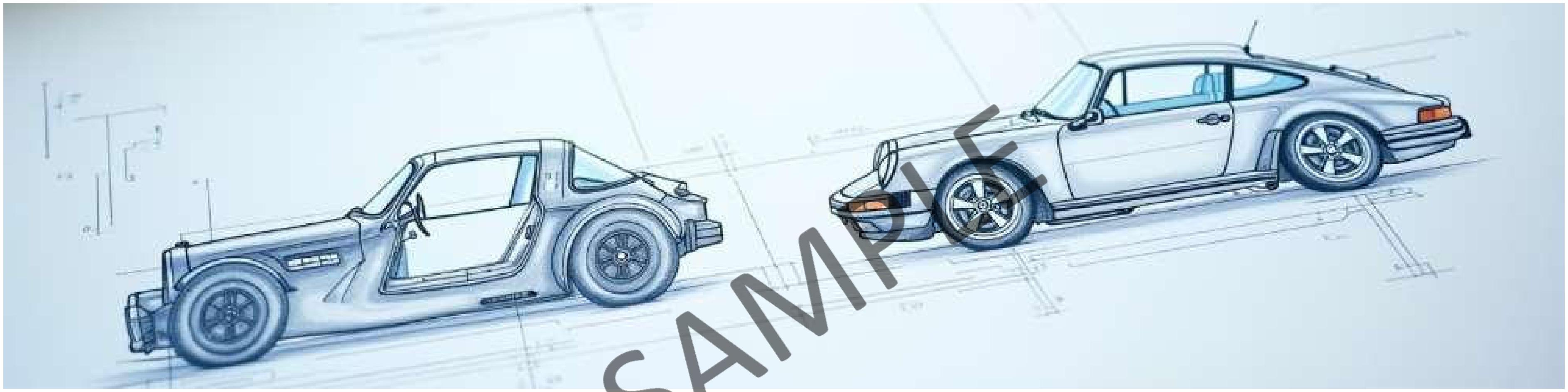
Table of Contents

Chapter 1: Introduction



1.1 Brief History of WinForms and WPF

1.1.1 Windows Forms (WinForms)

Windows Forms, commonly known as WinForms, was introduced by Microsoft in 2002 as part of the .NET Framework. It was designed as a successor to Visual Basic forms, providing a way for developers to create Windows applications using managed code. WinForms offered a simpler, more intuitive approach to building desktop applications compared to the then-prevalent Win32 API.

Key points about WinForms:

- Released with .NET Framework 1.0
- Based on the `System.Windows.Forms` namespace
- Utilizes a straightforward event-driven programming model
- Provides a WYSIWYG (What You See Is What You Get) designer in Visual Studio, allowing changes made visually to be immediately reflected in the application.

1.1.2 Windows Presentation Foundation (WPF)

Windows Presentation Foundation (WPF) was released by Microsoft in 2006 as part of .NET Framework 3.0. WPF represented a significant shift in Windows desktop application development, introducing a new rendering engine and a more flexible approach to UI design.

Key points about WPF:

- Introduced with .NET Framework 3.0
- Based on the `System.Windows` namespace

- Uses XAML (eXtensible Application Markup Language) for defining user interfaces
- Provides a clear separation between UI design and application logic
- Offers advanced graphics capabilities, including hardware acceleration

1.2 Purpose of the Guide

This guide serves as a comprehensive resource for developers looking to transition between WinForms and WPF. Whether you're familiar with WinForms and looking to dive into WPF or a WPF developer needing to manage legacy WinForms projects, this guide is designed to:

1. Highlight the similarities and differences between WinForms and WPF.
2. Provide a clear understanding of the core concepts and architecture of both frameworks.
3. Offer practical examples and best practices for working with each technology.
4. Guide developers in choosing the right framework for their specific needs.
5. Assist in migration strategies for moving between the two frameworks.

By comparing WinForms and WPF side-by-side, this guide will build on your existing knowledge while introducing new concepts and techniques. We will explore everything from basic UI design to advanced topics like data binding, styling, and performance optimization.

As we progress through the chapters, you'll gain insights into how each framework approaches common development tasks, allowing you to make informed decisions about which technology to use for your projects and how to effectively work with both when necessary.

1.3 Sample Code

The example projects and source code referenced in this book are based on .NET Framework 4.8 for WinForms and .NET Core 8.0 for WPF. We chose to use .NET Framework 4.8 for WinForms primarily because the WinForms designer for .NET 8.0 in Visual Studio 2022 is still inconsistent. Additionally, we considered the substantial user base and the number of applications still relying on .NET Framework 4.8 and earlier versions.

Sample projects are available for download from https://github.com/Code-Art-Engineering/FromWinFormsToWPF.

Chapter 5: UI Design

This chapter explores the different approaches to UI design in WinForms and WPF, highlighting the tools and methodologies used in each framework.

5.1 WinForms: Designer-centric Approach

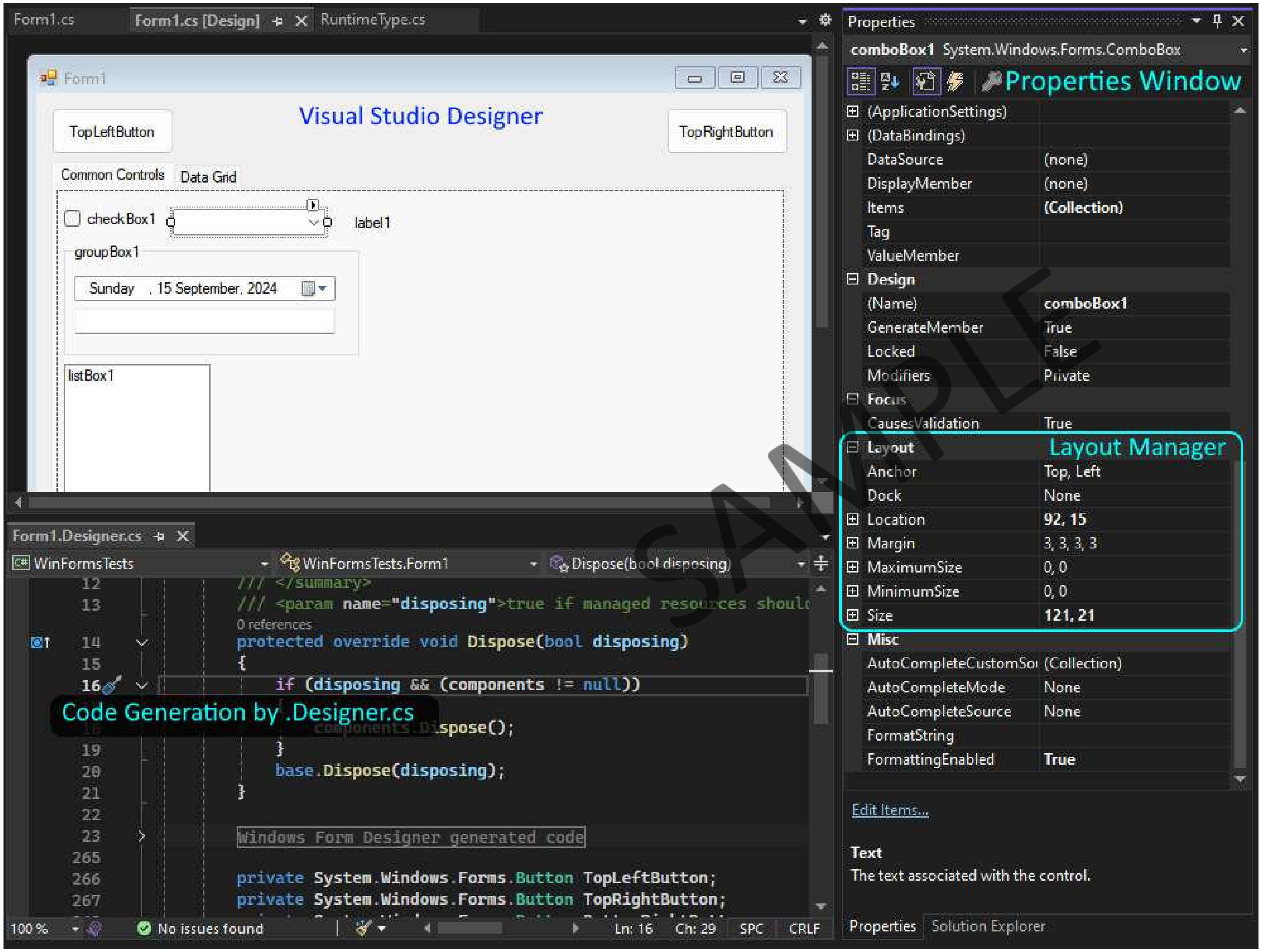
WinForms traditionally relies heavily on a visual designer integrated into development environments like Visual Studio for creating user interfaces. This designer allows developers to visually arrange controls like buttons, text boxes, and labels on a form using a drag-and-drop approach, providing a WYSIWYG (What You See Is What You Get) interface. It simplifies the process by auto-generating the underlying code, allowing developers to focus more on layout and design without writing extensive UI-related code manually. The designer also supports properties and event handling, streamlining the connection between the UI and business logic.

5.1.1 Key Features:

1. **Visual Studio Designer**: A WYSIWYG (What You See Is What You Get) interface for dragging and dropping controls to forms.
2. **Properties Window**: Allows easy modification of control properties without writing code.
3. **Layout Managers**: Docking, anchoring, and auto-sizing for basic responsive layouts.
4. **Code Generation**: The designer generates underlying code automatically.

5.1.2 Workflow:

1. Create a new Windows Forms project in Visual Studio.
2. Use the Toolbox to drag and drop controls onto the form.
3. Arrange controls using the mouse or properties window.
4. Set properties like size, color, and text in the Properties window.
5. Double-click controls to generate event handler methods.
6. Write code to handle events and implement functionality.



```
/// </summary>
/// <param name="disposing">true if managed resources shoulc
protected override void Dispose(bool disposing)
{
    if (disposing && (components != null))

        components.Dispose();
    }
    base.Dispose(disposing);
}

Windows Form Designer generated code

private System.Windows.Forms.Button TopLeftButton;
private System.Windows.Forms.Button TopRightButton;
```

comboBox1 System.Windows.Forms.ComboBox

⊞ (ApplicationSettings)	
⊞ (DataBindings)	
DataSource	(none)
DisplayMember	(none)
Items	**(Collection)**
Tag	
ValueMember	
⊟ **Design**	
(Name)	**comboBox1**
GenerateMember	True
Locked	False
Modifiers	Private
⊟ **Focus**	
CausesValidation	True
⊟ **Layout**	
Anchor	Top, Left
Dock	None
⊞ Location	**92, 15**
⊞ Margin	3, 3, 3, 3
⊞ MaximumSize	0, 0
⊞ MinimumSize	0, 0
⊞ Size	**121, 21**
⊟ **Misc**	
AutoCompleteCustomSo	(Collection)
AutoCompleteMode	None
AutoCompleteSource	None
FormatString	
FormattingEnabled	**True**

Text
The text associated with the control.

layout affecting flags
- Custom metadata classes: For specialized property behaviors

10.3.1.3 Property Value Resolution Hierarchy

One of the most powerful features of dependency properties is their sophisticated value resolution system. When you request a property value, the system evaluates multiple potential sources in a specific order of precedence:

1. **Local Value**: Values set directly on the object instance
2. **Triggered Style Values**: Values from style triggers that are currently active
3. **Template Values**: Values from the control's template
4. **Style Values**: Values from applied styles
5. **Inherited Values**: Values inherited from parent elements (for inheritable properties)
6. **Default Values**: The default value specified in property metadata

This hierarchy allows for flexible and predictable property value resolution. For example, a local value will always take precedence over a styled value, but both can coexist, and removing the local value will cause the styled value to become effective again.

The system also supports value coercion, where the final value can be modified by a coercion callback before being stored or returned. This enables scenarios like ensuring values stay within valid ranges or converting between related types.

10.3.2 Creating Your First Custom Dependency Property

Now that we understand the architectural foundation, let's create practical dependency properties with working examples.

10.3.2.1 Basic Dependency Property Declaration

The most straightforward dependency property declaration follows the standard pattern established by WPF. Here's a complete example of a custom control with a simple dependency property:

```
public class CustomTextBlock : Control
{
    // Dependency property declaration
    public static readonly DependencyProperty CustomTextProperty =
        DependencyProperty.Register(
            nameof(CustomText),
            typeof(string),
            typeof(CustomTextBlock),
            new PropertyMetadata(string.Empty, OnCustomTextChanged));

    // Property change callback
    private static void OnCustomTextChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)
    {
        var control = (CustomTextBlock)d;
        var newValue = (string)e.NewValue;
        var oldValue = (string)e.OldValue;
        // Custom logic when property changes
        control.OnCustomTextChanged(oldValue, newValue);
    }

    // Instance method for handling property changes
    protected virtual void OnCustomTextChanged(string oldValue, string newValue)
    {
        // Override in derived classes for custom behavior
        // Raise events, update UI, etc.
    }
    // CLR property wrapper (discussed in next section)
    public string CustomText
    {
        get => (string)GetValue(CustomTextProperty);
        set => SetValue(CustomTextProperty, value);
    }
}
```

This example demonstrates several important concepts:

- The static `DependencyProperty` field uses the naming convention of appending "Property" to the property name
- The `Register` method creates and returns the dependency property instance
- The property change callback is static and receives the changed object and event arguments
- An instance method provides a more convenient override point for derived classes

10.3.2.2 Property Wrappers and Best Practices

The CLR property wrapper is crucial for making dependency properties accessible through normal property syntax. However, there are important best practices to follow:

Correct Wrapper Implementation:

12.3 Quick Reference Table

WinForms	WPF Equivalent
`Refresh()`	`InvalidateVisual()`
`InvokeRequired`	`!Dispatcher.CheckAccess()`
`BeginInvoke`	`Dispatcher.BeginInvoke`
`DesignMode`	`DesignerProperties.GetIsInDesignMode(...)`
`System.Drawing`	`System.Windows.Media`
`Color.Black`	`Colors.Black`
`Color.FromArgb()`	`Color.FromRgb()`
`System.Windows.Forms`	`System.Windows.Controls`
`System.Drawing.Shapes`	`System.Windows.Shapes`
`PointF`/`RectangleF`	`Point`/`Rect`
`GraphicsPath`	`PathGeometry`
`Graphics.TranslateTransform`	`DrawingContext.PushTransform(...)`
`Font`	`Text`-related properties (e.g., `FontFamily`)

12.5 Case Study: NuGet Package Migration (CodeArtEng.Diagnostics)

Migrating from WinForms to WPF can be challenging, but a systematic approach can make the process much more manageable. This guide will provide a step-by-step explanation for migrating a project based on CodeArtEng.Diagnostics, offering detailed instructions to help ensure a smooth transition.

> **Reference Project**
>
> - **GitHub Link**: https://github.com/Code-Artist/CodeArtEng.Diagnostics
> - **WinForm Project Name**: CodeArtEng.Diagnostics
> - **WPF Project Name**: CodeArtEng.Diagnostics.WPF

12.5.1 Identify Generic and Platform Specific Class

1. First, identify generic classes and platform-specific classes (WinForm) from original WinForm projects. The attached class diagram highlights each class from WinForm project as either generic or platform-specific.