

Table of Contents

Chapter 1: Introduction	15
1.1 Brief History of WinForms and WPF	16
1.1.1 Windows Forms (WinForms)	16
1.1.2 Windows Presentation Foundation (WPF)	16
1.2 Purpose of the Guide	18
1.3 Sample Code	20
Chapter 2: Architecture and Concepts	21
2.1 WinForms: Event-driven, Controls-based Architecture	22
2.1.1 Key Architectural Points:	22
2.1.2 Core Concepts:	22
2.2 WPF: XAML-based, Separation of UI and Logic	23
2.2.1 Key Architectural Points:	23
2.2.2 Core Concepts:	23
2.3 Summary	24
Project Structures	25

3.1 Application Startup	26
3.1.1 WinForms (Program.cs)	27
3.1.2 WPF (App.xaml)	28
3.2 Forms	30
3.2.1 WinForms (Form)	30
3.3 WPF (Window)	35
3.3.1 UI Definition File (.xaml)	36
3.3.2 Code-Behind File (.xaml.cs)	37
3.4 Summary of Key Differences:	39
Chapter 4: Layout Management	40
4.1 WinForms: Absolute Positioning and Anchoring	42
4.1.1 Component Positioning	42
4.1.2 Anchor	43
4.1.3 Docking	44
4.1.4 TableLayoutPanel and FlowLayoutPanel	47
4.2 WPF: Layout Containers and Grid System	49
4.2.1 Layout Containers	49
4.2.2 HorizontalAlignment and VerticalAlignment	52
4.3 Common Layout Properties in Both WinForms and WPF	53

4.3.1 Margin and Padding	53
4.3.2 MinWidth, MaxWidth, MinHeight, MaxHeight	55
4.4 Example: Compare Positioning and Alignment in WinForms and WPF	56
4.4.1 Size and position	57
4.4.2 Alignment	59
4.5 Key Differences and Considerations	61
Chapter 5: UI Design	62
5.1 WinForms: Designer-centric Approach	63
5.1.1 Key Features:	63
5.1.2 Workflow:	63
5.2 WPF: XAML-centric Approach	65
5.2.1 Key Features:	65
5.2.2 Workflow:	65
5.3 Key Differences and Considerations	67
Chapter 6: MVVM in WinForms and WPF with Data Binding	69
6.1 MVVM Components	71
6.1.1 Model	71
6.1.2 View	71
6.1.3 ViewModel	72

6.2 MVVM in WPF	73
6.2.1 Simple WPF MVVM	73
6.2.2 Binding Modes in WPF	78
6.2.3 How It Works	80
6.2.4 Define Data Bindings from Designer	80
6.3 MVVM in WinForms	83
6.3.1 Simple WinForms MVVM	83
6.3.2 Binding Modes in WinForms	90
6.3.3 Summary	92
6.4 Conclusion	93
Chapter 7: Styling and Theming	94
7.1 Styling and Theming in WinForms	96
7.1.1 Styling Individual Controls	96
7.1.2 Application-wide Theming with Custom Controls	98
7.1.3 Global Styling with System-wide Colors	99
7.1.4 Managing Application-wide Styling through Resources	101
7.1.5 Third-party library	104
7.1.6 Summary	105
7.2 Styling and Theming in WPF	106

7.2.1 Styles	106
7.2.2 Templates	109
7.2.3 Template Binding from WPF Designer:	114
7.2.4 Resources	116
7.2.5 Theming in WPF	122
7.3 Conclusion	124
Chapter 8: Events and Commands	125
8.1 Event-Driven Programming in WinForms	126
8.1.1 Event Handlers in WinForms	126
8.1.2 Common WinForms Events	129
8.1.3 Custom Events in WinForms	129
8.2 Routed Events and Commands in WPF	133
8.2.1 Routed Events	133
8.2.2 The Command Pattern in WPF	136
8.2.3 Built-in WPF Commands	138
8.3 Conclusion	139
Chapter 9: Graphics and Animation	140
9.1 Graphics in WinForms	142
9.1.1 Basic Drawing with GDI+	142

9.1.2 Working with Images	143
9.2 Animation in WinForms	146
9.2.1 Flickering in WinForms	148
9.3 Graphics in WPF	152
9.3.1 Vector Graphics	152
9.3.2 3D Graphics	154
9.4 Animation in WPF	157
9.4.1 Property Animations	159
9.4.2 Keyframe Animations	165
9.4.3 Easing Functions	166
9.4.4 Key Points to Remember	169
9.5 Conclusion	170
Chapter 10: Dependency Properties	171
10.1 Introduction to Dependency Properties	172
10.1.1 What are Dependency Properties and Why They Exist	172
10.1.2 The Problem They Solve in WPF Architecture	172
10.1.3 Relationship to Regular CLR Properties	173
10.1.4 When to Use Dependency Properties vs Regular Properties	175
10.2 Key Features and Benefits	178

10.2.1 Property Value Inheritance	178
10.2.2 Data Binding Support	179
10.2.3 Property Change Notifications	180
10.2.4 Memory Efficiency Through Sparse Storage	181
10.2.5 Summary	183
10.3 Basic Implementation and Usage	184
10.3.1 The Dependency Property System Architecture	184
10.3.2 Creating Your First Custom Dependency Property	187
10.4 Common Pitfalls and Best Practices for Beginners	199
10.4.1 Common Mistakes When Starting with Dependency Properties	199
10.4.2 Essential Naming Conventions	199
10.4.3 When NOT to Use Dependency Properties	200
10.4.4 Debugging Dependency Property Issues	201
10.4.5 Basic Performance Considerations	202
10.4.6 Testing Dependency Properties	203
10.5 Property Metadata	204
10.5.1 PropertyMetadata vs FrameworkPropertyMetadata	204
10.5.2 Default Values and Property Changed Callbacks	209
10.5.3 Value Coercion	211

10.5.4 Validation Callbacks	213
10.5.5 Best Practices	215
10.6 Advanced Implementation Techniques (Further Reading)	216
10.6.1 Advanced Dependency Property Creation	216
10.7 Real-World Applications (Further Reading)	220
10.7.1 Custom Controls with Dependency Properties	220
10.7.2 Attached Behaviors	223
10.7.3 Integration with Data Binding and MVVM	226
10.7.4 Advanced Troubleshooting Techniques	230
10.7.5 Performance Anti-Patterns	232
10.7.6 Common Mistakes to Avoid	234
Chapter 11: Migration Strategies	240
11.1 To Migrate or Not	241
11.2 Understanding Key Differences in Components	243
11.2.1 Common or Smiliar WinForms and WPF Components	243
11.2.2 Unique WinForms Components	245
11.2.3 Unique WPF Components	246
11.3 A Practical Guide with Code Conversions	248
11.4 Quick Reference Table	249

11.5 Detailed Conversions with Examples	251
11.5.1 Refreshing the UI	251
11.5.2 Thread-Safe UI Updates	251
11.5.3 Detecting Design Mode	252
11.5.4 Colors and Namespaces	253
11.5.5 Replacing Controls and Shapes	254
11.5.6 Geometry and Drawing	254
11.5.7 Transforming Graphics	255
11.5.8 Font Handling	256
11.6 Final Tips	258
11.7 Case Study: NuGet Package Migration (CodeArtEng.Diagnostics)	259
11.7.1 Identify Generic and Platform Specific Class	259
11.7.2 Create WPF project	261
11.7.3 Migrate Generic Class	262
11.7.4 Configure WPF Project	265
11.7.5 Migrate Unit Test Project (Optional)	267
11.7.6 Migrate DiagnosticsTextBox Control	268
11.7.7 Create WPF Example Project	273
11.7.8 Testing and Performance Optimization	275

11.7.9 Summary	278
11.8 Best Practices for Migration	279
11.9 Conclusion	280
Chapter 11: Performance Considerations	281
11.10 WinForms: Lightweight and Fast Startup	282
11.10.1 Advantages	282
11.10.2 Considerations	282
11.10.3 Optimization Tips for WinForms	282
11.10.4 Recommended Use Cases	283
11.11 WPF: Hardware Acceleration and Complex UI Performance	285
11.11.1 Advantages	285
11.11.2 Considerations	285
11.11.3 Optimization Tips for WPF	285
11.11.4 Recommended Use Cases	286
11.12 Summary Comparison	287
Chapter 12: Migration Strategies	288
12.1 To Migrate or Not	289
12.2 Understanding Key Differences in Components	291
12.2.1 Common or Smiliar WinForms and WPF Components	291

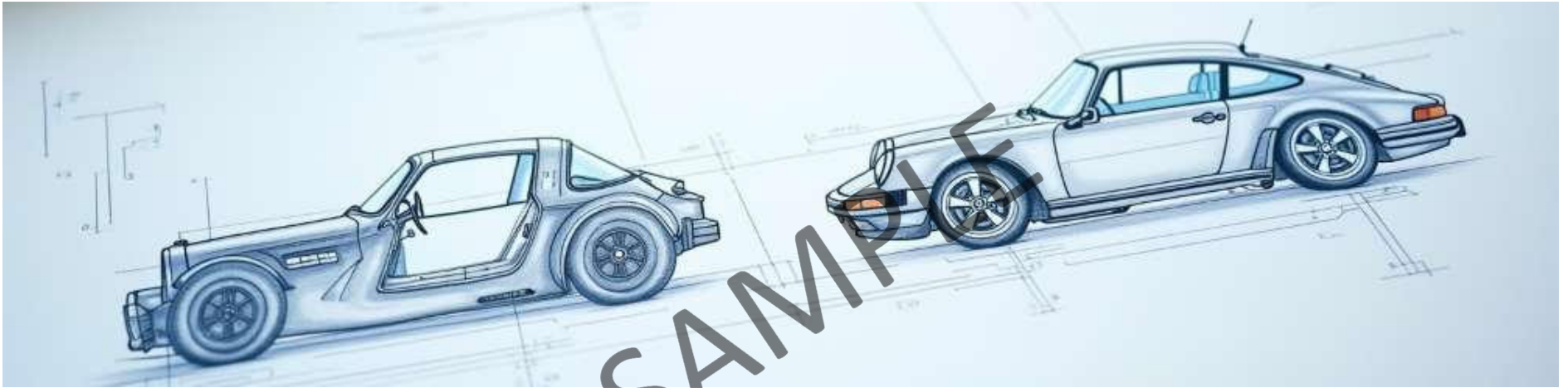
12.2.2 Unique WinForms Components	293
12.2.3 Unique WPF Components	294
12.3 Quick Reference Table	296
12.4 Detailed Conversions with Examples	298
12.4.1 Refreshing the UI	298
12.4.2 Thread-Safe UI Updates	298
12.4.3 Detecting Design Mode	299
12.4.4 Colors and Namespaces	300
12.4.5 Replacing Controls and Shapes	301
12.4.6 Geometry and Drawing	302
12.4.7 Transforming Graphics	302
12.4.8 Font Handling	303
12.4.9 Final Tips	304
12.5 Case Study: NuGet Package Migration (CodeArtEng.Diagnostics)	305
12.5.1 Identify Generic and Platform Specific Class	305
12.5.2 Create WPF project	307
12.5.3 Migrate Generic Class	308
12.5.4 Configure WPF Project	311
12.5.5 Migrate Unit Test Project (Optional)	313

12.5.6 Migrate DiagnosticsTextBox Control	314
12.5.7 Create WPF Example Project	319
12.5.8 Testing and Performance Optimization	321
12.5.9 Summary	324
12.6 Best Practices for Migration	325
12.7 Conclusion	326
Chapter 13: Visual Studio Settings	327
13.1 Application Icon	328
Chapter 14: Controls	330
14.1 Common Controls	331
14.1.1 General Properties	331
14.1.2 Button / Label	331
14.1.3 CheckBox	331
14.1.4 ComboBox	332
14.1.5 DateTimePicker / DatePicker	332
14.1.6 GroupBox	334
14.1.7 ListBox	334
Chapter 90: WinForms Common Issue and Solutions	335
90.1 Class not the first class in the File	336

Chapter 91: WPF Common Issue and Solution	337
91.1 Some assembly references are missing	338

SAMPLE

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

SAMPLE

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.

SAMPLE

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>.

SAMPLE

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

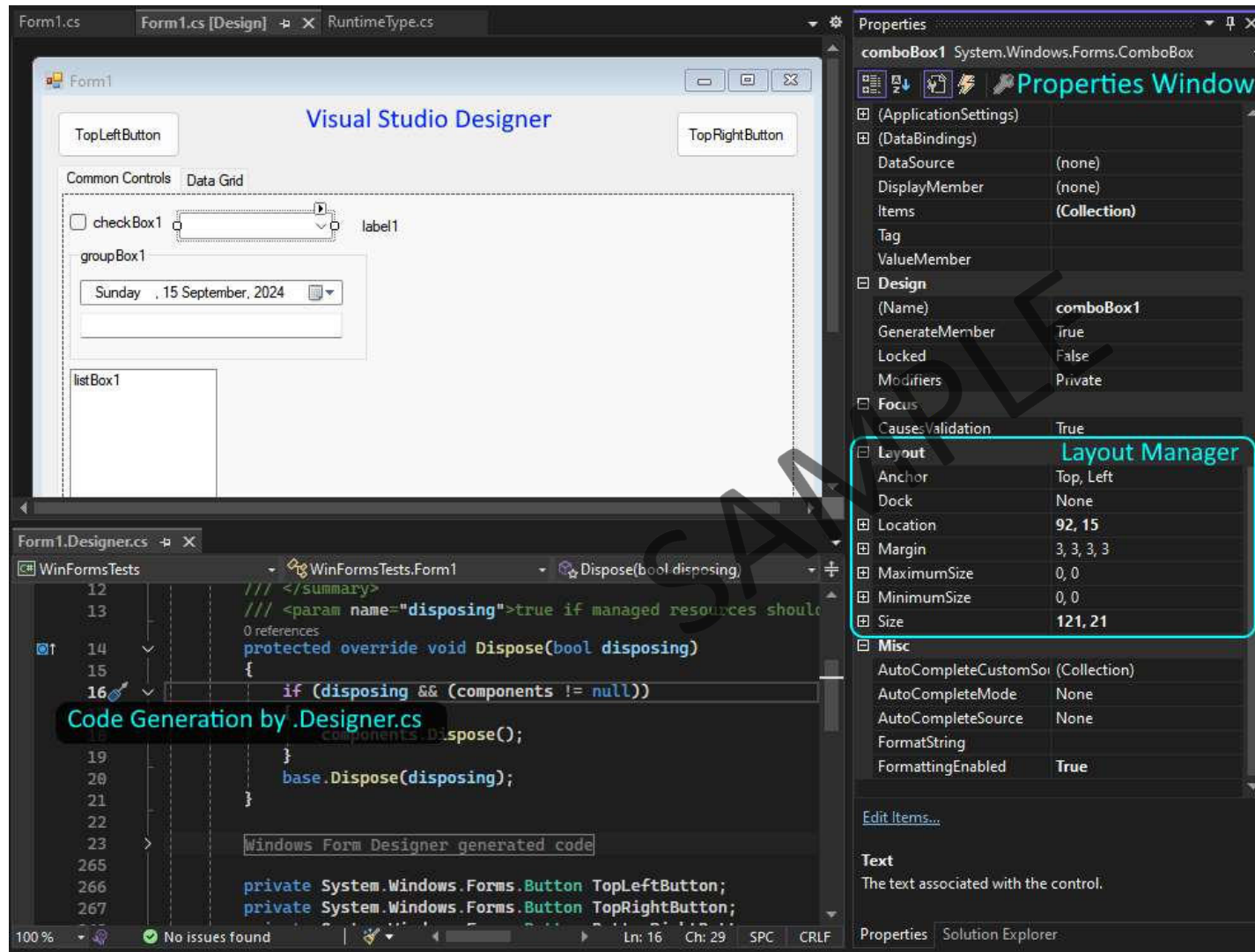
WinForms traditionally relies heavily on a visual designer integrated into development environments like Visual Studio for creating user interfaces. This designer allows developers to visually arrange controls like buttons, text boxes, and labels on a form using a drag-and-drop approach, providing a 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.



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., <code>FontFamily</code>)

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.