WPF-Textelemente für UI Automation sichtbar machen

Nov 5, 2026 · @Micha

Nicht-interaktive Elemente innerhalb eines `ControlTemplate` erscheinen in WPF nicht in der Control-View des UI-Automation-Baums. Testwerkzeuge finden sie deshalb nicht, obwohl sie auf dem Bildschirm stehen — typischerweise alle `TextBlock` in einem Zeilen-Template. Vier Änderungen an der Anwendung machen sie dauerhaft automatisierbar.

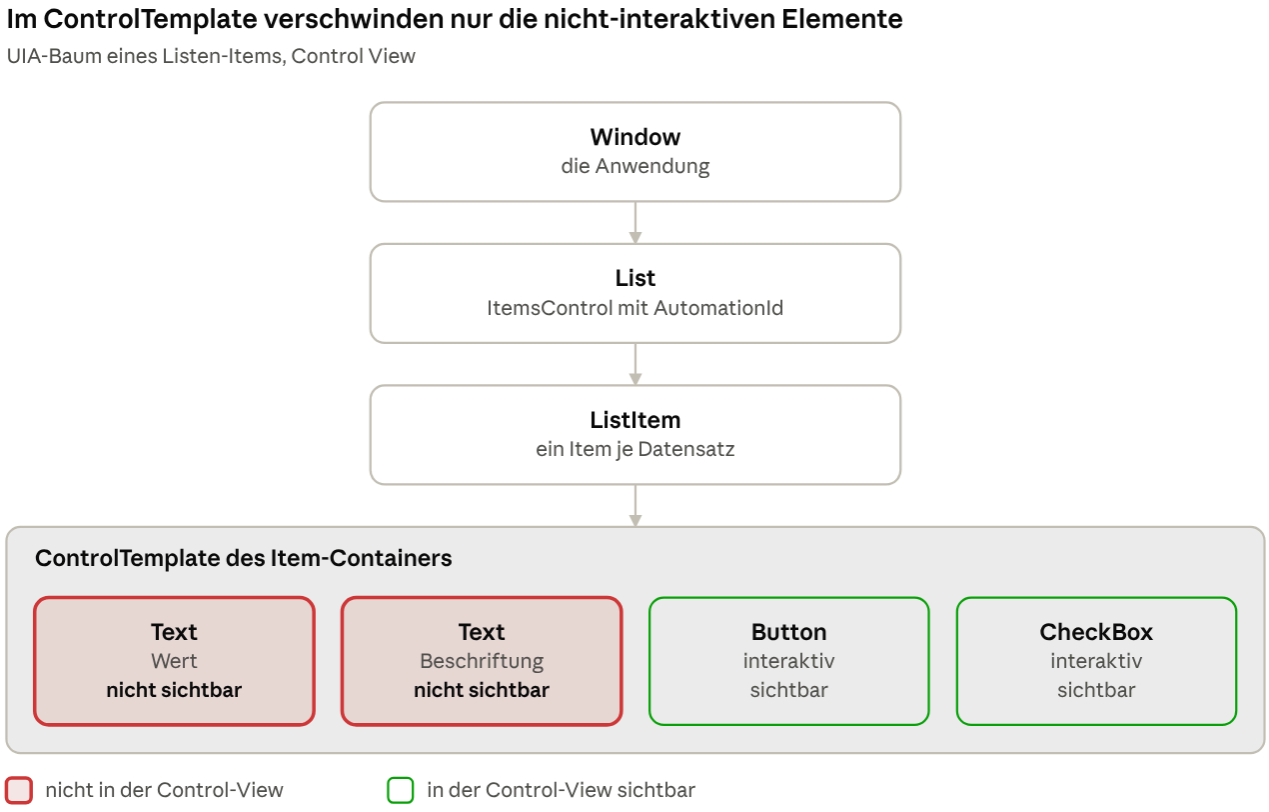
Überblick

Die Änderungen betreffen nur XAML und eine kleine Hilfsklasse. Fachlogik, Layout und Aussehen der Anwendung bleiben unberührt.

Änderung	Wofür sie sorgt	Notwendigkeit
1. Container identifizierbar machen	ein Item der Liste eindeutig ansprechen	Pflicht
2. Textelemente in die Control-View holen	die Werte im Template überhaupt finden	Pflicht
3. Virtualisierung entscheiden	auch nicht sichtbare Items erreichen	Pflicht
4. Eigene Controls mit AutomationPeer versehen	Custom Controls bedienbar machen	Nur bei Bedarf

Schritt 1 und 2 gehören zusammen: ohne eindeutigen Container ist ein gefundenes Textelement keinem Item zuzuordnen, ohne Schritt 2 ist es überhaupt nicht zu finden.

Ursache



UIA-Baum eines Listen-Items · 2 von 4 Elementen fehlen

Seit .NET Framework 4.8 nimmt WPF nicht-interaktive Elemente aus der Control-View des UIA-Baums heraus, wenn sie Teil eines `ControlTemplate` sind. Gedacht ist das als Entlastung für Screenreader: Template-Interna sollen nicht vorgelesen werden.

Betroffen sind damit praktisch alle `TextBlock` in einem Zeilen-Template — also genau die Werte, die ein Test lesen will. Interaktive Elemente im selben Template bleiben Control-Elemente; daran lässt sich der Effekt gut gegenprüfen.

Für die Testautomatisierung ist das entscheidend, weil die Elementsuche der meisten Werkzeuge über die Control-View läuft. In der Raw-View sind die Texte vorhanden, doch die ist für Tests kaum brauchbar: sie führt jedes Border und jeden ContentPresenter mit und sieht nach jeder Layoutänderung anders aus.

Zwei Gründe für ein fehlendes Element

Beim Prüfen mit Inspect.exe begegnen zwei Fälle, die gleich aussehen, aber verschiedene Ursachen und verschiedene Abhilfen haben.

Fall	Im UIA-Baum	Abhilfe
Kein Control-Element	vorhanden, aber nur in der Raw-View	AutomationPeer, siehe Schritt 2
Ausgeblendet	gar nicht vorhanden	keine — korrektes Verhalten

Der erste Fall ist der aus dem vorigen Abschnitt und der Gegenstand dieser Anleitung.

Der zweite ist kein Fehler: `Visibility="Collapsed"` oder `"Hidden"` entfernt ein Element vollständig aus dem UIA-Baum, und das ist richtig so — was nicht zu sehen ist, soll auch nicht automatisierbar sein. Betroffen ist alles, was an einem `Visibility`-Binding, einem `DataTrigger` oder einem Converter hängt. In einem Zeilen-Template trifft das häufig ganze Blöcke, die nur für bestimmte Datensätze eingeblendet werden.

Faustregel: Ein Element, das in Inspect.exe überhaupt nicht auftaucht — auch nicht in der Raw-View — ist ausgeblendet, nicht gefiltert. Prüfen Sie dann zuerst die Visibility-Bindung, bevor Sie einen AutomationPeer bauen.

Und die Gegenprobe für den ersten Fall: Interaktive Elemente im selben Template — ein `Button`, eine `CheckBox`, eine `ComboBox` — bleiben Control-Elemente und sind erreichbar, solange sie eingeblendet sind. Findet Ihr Werkzeug die Schaltfläche, aber nicht die Beschriftung daneben, ist das der Filtermechanismus und kein Fehler in Ihrem XAML.

Schritt 1 — Den Container identifizierbar machen

Jedes Item einer `ListView`, `ListBox` oder eines `ItemsControl` entsteht aus demselben `ControlTemplate`. Die Namen darin sind deshalb in jedem Item identisch. Ein Testwerkzeug findet also so viele gleich benannte Elemente, wie Items angezeigt werden, und kann sie nicht auseinanderhalten.

Der Container selbst — `ListViewItem`, `ListBoxItem` — ist bereits ein Control-Element und für die Automatisierung sichtbar. Ihm fehlt nur ein eindeutiger Name. Zwei Setter im `ItemContainerStyle`, gebunden ans ViewModel, genügen:

```
<Style x:Key="_styleItemContainer" TargetType="{x:Type ListViewItem}">
  <Setter Property="AutomationProperties.AutomationId" Value="{Binding Id}"/>
  <Setter Property="AutomationProperties.Name"         Value="{Binding Id}"/>

  <!-- die übrigen Setter unverändert -->
</Style>
```

Wählen Sie dafür eine Eigenschaft, die den Datensatz eindeutig macht und sich **nicht** mit der Oberflächensprache ändert — eine fachliche Nummer, einen Schlüssel, einen Dateinamen. Ein übersetzter Anzeigetext ist ungeeignet, weil der Testpfad dann bei jeder Sprachumstellung bricht.

Ohne diesen Schritt bleibt nur der Index, also `ListItem[3]`, und der bricht, sobald sich Sortierung oder Filter ändern. Verlassen Sie sich auch nicht auf den `Name` des Containers: WPF leitet ihn aus `ToString()` des Datenobjekts ab, und der ist häufig für alle Items gleich.

Schritt 2 — Textelemente in die Control-View holen

Das ist der Kern. Die Eigenschaft `IsControlElement` lässt sich **nicht** über ein XAML-Attribut setzen — dazu braucht es einen eigenen AutomationPeer. `AutomationProperties.AutomationId` und `AutomationProperties.Name` allein genügen also nicht.

Schnellprobe vor dem Umbau

Zum Bestätigen des Befunds genügt ein Kompatibilitätsschalter in der `app.config` (bei .NET Core/5+ in der `runtimeconfig.json`):

```
<configuration>
  <runtime>
    <AppContextSwitchOverrides
value="Switch.UseLegacyAccessibilityFeatures.3=true" />
  </runtime>
</configuration>
```

Tauchen die Texte danach in der Control-View auf, ist die Ursache belegt. Als Dauerlösung ist der Schalter ungeeignet: er holt global **alle** Template-Interna zurück, bläht den UIA-Baum stark auf und bremst jede Suche.

Die Peer-Klasse

Einmal geschrieben, danach überall verwendbar:

```
using System;
using System.Windows.Automation.Peers;
using System.Windows.Automation.Provider;
using System.Windows.Controls;

namespace YourApp.Controls
{
```

```
    /// <summary>
    /// TextBlock, der auch innerhalb eines ControlTemplate in der
    /// UIA-Control-View sichtbar ist und seinen Text über das
    /// ValuePattern bereitstellt.
    /// </summary>
    public class AutomationTextBlock : TextBlock
    {
        protected override AutomationPeer OnCreateAutomationPeer()
            => new AutomationTextBlockPeer(this);
    }

    public class AutomationTextBlockPeer : TextBlockAutomationPeer,
IValueProvider
    {
        public AutomationTextBlockPeer(TextBlock owner) : base(owner) { }

        protected override bool IsControlElementCore() => true;
        protected override bool IsContentElementCore() => true;

        public override object GetPattern(PatternInterface patternInterface)
            => patternInterface == PatternInterface.Value
                 ? this
                 : base.GetPattern(patternInterface);

        bool   IValueProvider.IsReadOnly => true;
        string IValueProvider.Value      => ((TextBlock)Owner).Text;
        void   IValueProvider.SetValue(string value)
            => throw new InvalidOperationException();
    }
}
```

Das XAML

Tauschen Sie nur die **Wert**-TextBlocks aus. Beschriftungen, deren Text übersetzt wird, bleiben wie sie sind — sie taugen ohnehin nicht als Suchkriterium.

```
xmlns:a="clr-namespace:YourApp.Controls"

<a:AutomationTextBlock AutomationProperties.AutomationId="tbStatus"
                       AutomationProperties.Name="Status"
                       Style="{StaticResource _styleValueText}"
                       Text="{Binding Status}" />
```

Templates, die mehrfach verwendet werden

Wird ein `DataTemplate` an mehreren Stellen eingesetzt — etwa als `ContentTemplate` mehrerer `ContentControl` —, tragen alle Instanzen dieselbe AutomationId. Geben Sie dann zusätzlich jedem Host eine eigene Kennung:

```
<ContentControl AutomationProperties.AutomationId="ccLastChange"
                Content="{Binding LastChange}"
                ContentTemplate="{StaticResource _tmplTimestamp}" />
```

Der Testpfad läuft dann über drei Stufen: Container → Host → Element.

Was sich dadurch ändert

Sobald `AutomationProperties.Name` gesetzt ist, liefert die UIA-Eigenschaft `Name` nicht mehr den Anzeigetext, sondern den gesetzten Namen. Der Wert kommt dann über das ValuePattern, das die Peer-Klasse bereitstellt.

Lassen Sie `AutomationProperties.Name` weg, bleibt der Anzeigetext in `Name` — dann ist der Text direkt lesbar, die Suche nach dem Element aber sprachabhängig. Beides ist vertretbar; entscheiden Sie es einmal für die ganze Anwendung und halten Sie sich daran.

Wenn Tests Werte zwischenspeichern, sollte der Peer bei Textänderung zusätzlich `RaisePropertyChangedEvent` für `ValuePatternIdentifiers.ValueProperty` auslösen. Ohne das lesen cachende Clients den alten Stand.

Schritt 3 — Virtualisierung entscheiden

`ListView` und `ListBox` virtualisieren standardmäßig. Items außerhalb des sichtbaren Bereichs existieren im UIA-Baum überhaupt nicht. Auch nach Schritt 1 und 2 findet ein Test also nur, was gerade angezeigt wird.

Option	Änderung	Wirkung
Virtualisierung aus	`VirtualizingStackPanel.IsVirtualizing="False"` am ItemsControl	alle Items jederzeit erreichbar; bei langen Listen merklich langsamer beim Öffnen
Virtualisierung an	keine	der Test muss das Item vorher in den sichtbaren Bereich scrollen

Bei kurzen Listen ist das Abschalten der einfachere Weg. Bei mehreren hundert Einträgen lassen Sie die Virtualisierung besser an — das ist kein Hindernis, das Testwerkzeug scrollt das Item dann selbst heran, über das `ScrollItemPattern`.

Wichtig ist vor allem, dass die Entscheidung bewusst fällt: sie bestimmt, wie die Tests geschrieben werden müssen.

Schritt 4 — Eigene Controls mit AutomationPeer versehen

Custom Controls, die `OnCreateAutomationPeer` nicht überschreiben, erscheinen als strukturloses `Custom`-Element ohne Pattern — sichtbar, aber nicht bedienbar.

Betroffen sind vor allem:

- Eingabefelder — brauchen ein `ValuePattern`, damit ein Test Text setzen und zurücklesen kann
- Navigations- und Pfadleisten — die einzelnen Einträge sollten als eigene Peers mit `ControlType.Button` erscheinen
- zusammengesetzte Anzeigen, deren Teilwerte einzeln geprüft werden sollen

Für ein eingabefähiges Control leitet man üblicherweise von `TextBoxAutomationPeer` ab oder implementiert `IValueProvider` wie in Schritt 2. Für reine Anzeige-Controls genügt meist `FrameworkElementAutomationPeer` mit einem passenden `GetAutomationControlTypeCore`.

Dieser Schritt ist nur nötig, soweit die Tests diese Bedienelemente tatsächlich ansprechen sollen.

Ergebnis prüfen

Dafür braucht es kein Testwerkzeug. **Inspect.exe** aus dem Windows SDK genügt, alternativ **Accessibility Insights for Windows**.

1. Inspect.exe starten und den Modus auf **UI Automation** stellen.
2. Die Ansicht auf **Control View** umstellen — ausdrücklich nicht Raw View, denn dort waren die Elemente auch vorher schon zu sehen.
3. Ein Wertfeld in der Anwendung anklicken und die Eigenschaftsliste vergleichen.

Eigenschaft	Erwartet nach der Änderung
`IsControlElement`	`true`
`ControlType`	`Text`
`AutomationId`	die vergebene Kennung
`Name`	der gesetzte Name, sonst der Anzeigetext
Patterns	`ValuePattern` vorhanden, `Value` = angezeigter Text
`AutomationId` des Containers	die fachliche Kennung aus Schritt 1

Vor der Änderung fehlt das Element in der Control View, und `IsControlElement` steht auf `false`.

Ist die Tabelle erfüllt, ist das Feld über einen stabilen zweistufigen Pfad erreichbar:

```
//List[@AutomationId="<Liste>"]/ListItem[@AutomationId="
<Kennung>"]/Text[@AutomationId="tbStatus"]
```

Dieser Pfad überlebt Sprachumstellungen, Sortieränderungen und Wertwechsel. Die Virtualisierung bleibt zu beachten, solange sie nicht abgeschaltet ist.