Inhaltsverzeichnis

Tipps und Tricks

Hier finden Sie verschiedene Tipps und Tricks rund um C#, .NET und Visual Studio (Verschieden Versionen).

ContextSwitchDeadlock erkennen und umgehen

Fehlermeldung Visual Studio 2019

Message
ContextSwitchDeadlock wurde erkannt. Message: Die CLR konnte 60 Sekunden lang keinen Übergang vom COM-Kontext 0x2c32f90 zum COM-Kontext 0x2c331e0 durchführen. Der Thread, der Besitzer des Zielkontexts/-apartments ist, wartet entweder, ohne Meldungen zu verschieben, oder verarbeitet eine äußerst lang dauernde Operation, ohne Windows-Meldungen zu verschieben. Eine solche Situation beeinträchtigt in der Regel die Leistung und kann sogar dazu führen, dass die Anwendung nicht mehr reagiert oder die Speicherauslastung immer weiter zunimmt. Zur Vermeidung dieses Problems sollten alle STA-Threads (Singlethread-Apartment) primitive Typen verwenden, die beim Warten Meldungen verschieben (z.B. CoWaitForMultipleHandles), und bei lange dauernden Operationen generell Meldungen verschieben.

Diese tritt beim abfragen von Fenstertitel der Anwendungen auf, der Code dazu:

```
 1. public string Text
 2.     {
 3.      get
 4.     {
 5.      try
 6.     {
 7.      StringBuilder title = new StringBuilder(260, 260);
 8.      UnManagedMethods.GetWindowText(this.hWnd, title, title.Capacity);
 9.      return title.ToString();
10.     }
11.      catch{return "";}
12.     }
13.     }
14.
15. private class UnManagedMethods
16.     {
17.      [DllImport("user32", CharSet = CharSet.Auto)]
18.      public extern static int GetWindowText(IntPtr hWnd, StringBuilder
    lpString,     int cch);
19.      ...
20.     }
```

Der code wird in Visual Studio 2019 im Debug Modus ausgeführt. Wie kann man dieses „hängen bleiben“ erkennen und abbrechen, gibt es da überhaupt eine Möglichkeit?

Erklärung

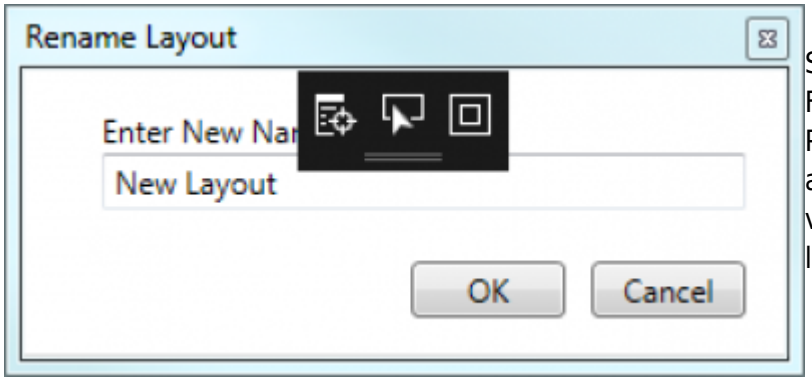
Wenn man im Debug Modus anhält, dann werden auch keine Windows-Nachrichten mehr verarbeitet. Das heißt, die COM-Komponente verarbeitet eine Windows-Nachricht, die verursacht, dass in deinen Code gesprungen wird. Sollte dann binnen 60 Sekunden keine Rückantwort kommen, dann erhälst Du diese Fehlermeldung, weil die COM Komponente keine weiteren Nachrichten verarbeiten kann derweil.

Lösung

Einfach die Exception in den Visual Studio Einstellungen abschalten.

In Visual Studio 2019 zusätzliche Debug Informationen auschalten

Zusätzliche Debug Informationen



Die Standardeinstellungen in Visual Studio 2019 zeigt oben auf jedem WPF Fenster zusätziche Tool zum debugen des Programm an. Nachteil ist, dass damit auch darunterliegende Komponenten verdeckt werden. Mit folgenden Schritten lässt sich das auch ausschalten:

English Version : Tools → Options → Debugging → General → Enable UI Debugging Tools for XAML
Deutsche Version : Extras → Optionen → Debugging → Allgemein → UI-Debugtool für XAML aktivieren

Setzen oder entfernen Sie einfach das Häckchen.

Window und Screen Mouse Koordinaten ermitteln

Wie wir alle wissen gibt es Methoden die uns die Mausposition relativ zu anderen controls zurückgibt. Doch manchmal möchte wir auch die Mausposition ausserhalb des Fensters wissen. Diese Kurzanleitung soll einen kleinen Tipp sein.

```
1. #Mit folgenden zwei Methoden lässt sich die Mausposition relativ zu
   einem control ermitteln:
```

```
2. Mouse.GetPosition(IInputElement relativeTo)
3. MouseEventArgs.GetPosition(IInputElement relativeTo).
```

Demo Code

Bei diesem Beispiel wird die Mausposition auf der obersten Titelleiste (WindowTitle) angezeigt. Die Koordination sind innerhalb des Fensters auf die Zeichnungsfläche bezogen und ausserhalb des Fensters werden die Screen Koordinaten angezeigt.

```
1. namespace CoreLoader.Views
2.
3.
4. {
5.     /// <summary>
6.     /// Interaction logic for Main.xaml
7.     /// </summary>
8.     public partial class Main : Window
9.     {
10.         public Main(object datacontex)
11.         {
12.             InitializeComponent();
13.             DataContext = datacontex;
14.
15.             CompositionTarget.Rendering += OnRendering;
16.         }
17.
18.         private void OnRendering(object sender, EventArgs e)
19.         {
20.             var x = Mouse.GetPosition(this).X;
21.             var y = Mouse.GetPosition(this).Y;
22.             this.Title = Math.Round(y, 0).ToString() + " | " +
    Math.Round(x, 0).ToString();
23.         }
24.     }
25. }
```

Der Event **OnRendering()** wird vor dem Zeichnen des WPF Fenster ausgeführt. Dieses Beispiel ist eine verkürzte Abschrift und wurde zur Sicherung kopiert.

WPF, DependencyProperty.Register() or .RegisterAttached

English	Deutsch
I assume you meant DependencyProperty.Register and DependencyProperty.RegisterAttached. DependencyProperty.Register is used to register normal DependencyProperty. You can see those as just regular properties, with the added twist that they can take part in WPF's DataBinding, animations etc. In fact, they are exposed as normal property (with the get and set accessors) on top of the untyped DependencyObject.SetValue / GetValue. You declare those as part of your type. Attached properties on the other hand are different. They are meant as an extensibility system. If you have ever used Extenders in Windows Forms, they are kind of similar. You declare them as part of a type, to be used on another type. They are used a lot for layout-related information. For example, Canvas needs Left/Top coordinates, Grid needs a Row and a Column, DockPanel needs a Dock information etc. It would be a mess if all of this had to be declared on every Control that can be layouted. So they are declared on the corresponding panel, but used on any Control. You can use the same thing to attach any information to a DependencyObject if you need to. It can come in handy to just declare a piece of information that you can set in xaml just to be used later in a style for an existing class for example. So those two kind of DependencyProperty serve a very different purpose. Regular properties (registered through Register()) are used just like normal properties as part of the interface of your type. Attached properties (registered through RegisterAttached()) are used as an extensibility point on existing classes. Hope that clarifies it a bit.	Ich nehme an, Sie meinten DependencyProperty.Register und DependencyProperty.RegisterAttached. DependencyProperty.Register wird verwendet, um normale DependencyProperty zu registrieren. Sie können diese als ganz normale Eigenschaften betrachten, mit dem zusätzlichen Vorteil, dass sie an WPFs DataBinding, Animationen usw. teilnehmen können. In der Tat sind sie als normale Eigenschaft (mit den Get- und Set-Accessoren) auf dem untypisierten DependencyObject.SetValue / GetValue ausgesetzt. Sie deklarieren diese als Teil Ihres Typs. Angehängte Eigenschaften hingegen sind anders. Sie sind als ein System zur Erweiterung gedacht. Wenn Sie schon einmal Extender in Windows Forms verwendet haben, sind sie sehr ähnlich. Sie werden als Teil eines Typs deklariert, um in einem anderen Typ verwendet zu werden. Sie werden häufig für layoutbezogene Informationen verwendet. Zum Beispiel braucht Canvas Links/Oben-Koordinaten, Grid braucht eine Row und eine Column, DockPanel braucht eine Dock-Information usw. Es wäre unübersichtlich, wenn all dies für jedes Steuerelement, das für das Layout verwendet werden kann, deklariert werden müsste. Also werden sie auf dem entsprechenden Panel deklariert, aber auf jedem Control verwendet. Sie können dasselbe tun, um beliebige Informationen an ein DependencyObject anzuhängen, wenn Sie es brauchen. Es kann sehr nützlich sein, eine Information zu deklarieren, die man in xaml einstellen kann, um sie später in einem Stil für eine bestehende Klasse zu verwenden, zum Beispiel. Diese beiden Arten von DependencyProperty dienen also einem sehr unterschiedlichen Zweck. Reguläre Eigenschaften (registriert durch Register()) werden wie normale Eigenschaften als Teil der Schnittstelle Ihres Typs verwendet. Angehängte Eigenschaften (registriert durch RegisterAttached()) werden als Erweiterungspunkt für bestehende Klassen verwendet. Ich hoffe, das macht es ein wenig klarer.

--Denis Troller

How to Test Your Internal Classes in C# (NUnit)

How to test internal classes? (microsoft,) 2019-12-10 by Johnny Graber

One of the most important concepts of object-oriented design is encapsulation. You try to hide all the internal things of a class from the other developers and only offer them a subset of functionality to use. You can achieve this by setting an appropriate access modifier for your methods and classes:

- **public:** The type or member can be accessed by any other code in the same assembly or another assembly that references it.
- **private:** The type or member can be accessed only by code in the same class or struct.
- **protected:** The type or member can be accessed only by code in the same class, or in a class that is derived from that class.
- **internal:** The type or member can be accessed by any code in the same assembly, but not from another assembly.
- **protected internal:** The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly. (as in protected OR internal)
- **private protected:** The type or member can be accessed only within its declaring assembly, by code in the same class or in a type that is derived from that class. (as in private OR protected)

Public and private are the two most used access modifiers. You find them in all the examples, they are straight forward to use and do exactly what you expect. They are a great help to manage access to the methods in your classes and the classes themselves.

If we look at bigger parts of our application, we use code from different assemblies or NuGet packages. Those distribution formats have their own boundaries that you can use to enforce encapsulation. Public and private access modifiers are again a great help. However, over the years I appreciated the internal access modifier more and more.

Benefits of the internal access modifier

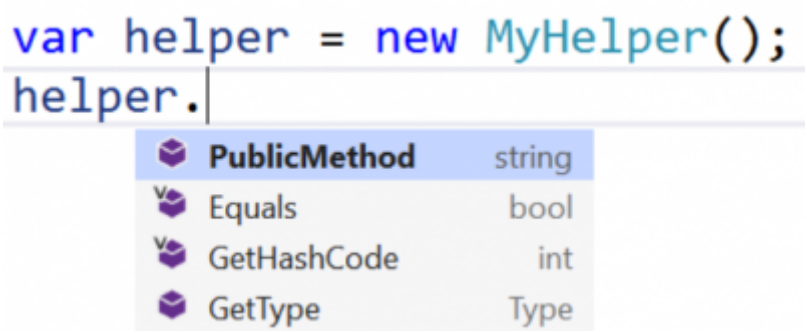
There is always that code that you need but has no place to go. It is not a class on its own and it does not fit to any other. At some point you stop searching for the right place and put it into a class called MyHelper. That code can’t be private, then many of your classes need them. And you do not want to make it public, then this code should not be called from outside your assembly.

The internal access modifier is exactly made for such use cases. By declaring the class or just a few methods as internal, you can access them from everywhere in your assembly but not from outside. All you need to do is to write internal instead of public or private:

```
1. public class MyHelper
2. {
3.     internal string InternalMethod()
4.     {
5.         return "should only be visible to the class itself & tests";
6.     }
7.
8.     public string PublicMethod()
9.     {
10.        return "Everyone can call this method";
11.    }
12.
```

```
13.     private string PrivateMethod()
14.     {
15.         return "you should not be able to call this directly";
16.     }
17. }
```

The users of your assembly or NuGet package do not know that this helper method exist. That allows you to freely move that code around to a better location or refactor it until you find a more fitting abstraction. All that without the need to change code outside your assembly – then no one else can call it directly.



How to test internal methods and classes?

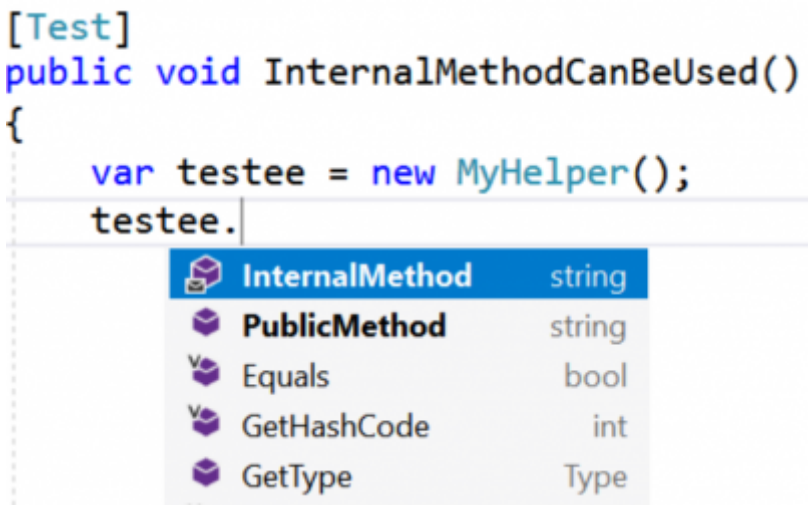
That helper code you marked with internal is most often important. Therefore, you should write extensive tests for those classes and methods. But how can you do that when you can't access that code from outside your assembly?

The .NET Framework offers the InternalsVisibleTo attribute to specify which other assemblies can access the internal methods and classes inside this assembly. All you need to do is to add this attribute to the AssemblyInfo.cs file and pass the name of your test assembly to the constructor:

```
1. [assembly: InternalsVisibleTo("Logic.Tests")]
```

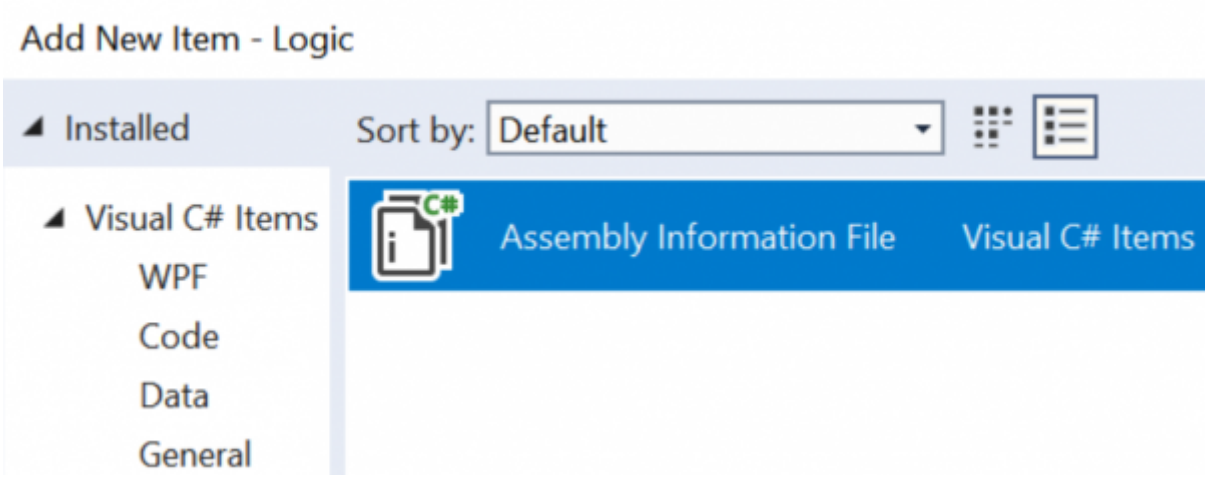
When you put this attribute to the AssemblyInfo.cs file, then all internal methods can be accessed by code inside the Logic.Tests assembly. To test your internal code this behaviour is exactly what you want. If this is too much, you can add this attribute in a specific class and only allow access to the internal methods of this class.

As soon as you recompile your assembly, the code in your test assembly can access your internal methods:



.Net Core

In .Net Core you do not have an AssemblyInfo.cs file. You can add one with the Add New Item dialog and set the attribute there in the same way you would do that in the .Net Full Framework and get exactly the same benefits.



.Net Standard project

As pointed out by Miguel Alho in the comments, you can add an ItemGroup in your *.csproj file to get the same effect. For that, paste this code as the last block before the closing project tag:

```
1. <ItemGroup>
2.   <AssemblyAttribute
   Include="System.Runtime.CompilerServices.InternalsVisibleTo">
3.     <_Parameter1>Logic.Tests</_Parameter1>
4.   </AssemblyAttribute>
5. </ItemGroup>
```

Conclusion

Use the internal access modifier the next time you have helper code that you need but no one else should call. This little keyword will help you to hide your mess inside your assembly and still allows you to write tests. With internal you get the best of both worlds without breaking encapsulation.

2019-12-10 by Johnny Graber

Working with Checkboxes in the WPF TreeView / Arbeiten mit Kontrollkästchen in der WPF TreeView

Introduction

This article reviews a WPF TreeView whose items contain checkboxes. Each item is bound to a ViewModel object. When a ViewModel object's check state changes, it applies simple rules to the check state of its parent and child items. This article also shows how to use the attached behavior concept to turn a TreeViewItem into a virtual ToggleButton, which helps make the TreeView's keyboard interaction simple and intuitive.

This article assumes that the reader is already familiar with data binding and templates, binding a TreeView to a ViewModel, and attached properties.

Background

It is very common to have a TreeView whose items are checkboxes, such as when presenting the user with a hierarchical set of options to select. In some UI platforms, such as WinForms, the standard TreeView control offers built-in support for displaying checkboxes in its items. Since element composition and rich data binding are two core aspects of WPF, the WPF TreeView does not offer intrinsic support for displaying checkboxes. It is very easy to declare a CheckBox control in a TreeView's ItemTemplate and suddenly every item in the tree contains a

Einführung

Dieser Artikel beschreibt eine WPF TreeView, deren Elemente Kontrollkästchen enthalten. Jedes Element ist an ein ViewModel Objekt gebunden. Wenn sich der Prüfstatus eines ViewModel-Objekts ändert, wendet es einfache Regeln auf den Prüfstatus seiner übergeordneten und untergeordneten Elemente an. Dieser Artikel zeigt auch, wie man das angehängte Verhaltenskonzept verwenden kann, um ein TreeViewItem in einen virtuellen ToggleButton zu verwandeln, der hilft, die Tastaturinteraktion des TreeViews einfach und intuitiv zu gestalten.

Dieser Artikel geht davon aus, dass der Leser bereits mit Datenbindung und Templates, der Bindung eines TreeViews an ein ViewModel und angehängten Eigenschaften vertraut ist.

Hintergrund

Es ist sehr üblich, einen TreeView zu haben, dessen Elemente Kontrollkästchen sind, z.B. wenn dem Benutzer ein hierarchischer Satz von Optionen zur Auswahl präsentiert wird. In einigen UI-Plattformen, wie z.B. WinForms, bietet das Standard-TreeView-Steuerelement integrierte Unterstützung für die Anzeige von Kontrollkästchen in seinen Elementen. Da Elementkomposition und reichhaltige

CheckBox. Add a simple {Binding} expression to the IsChecked property, and suddenly the check state of those boxes is bound to some property on the underlying data objects. It would be superfluous, at best, for the WPF TreeView to have an API specific to displaying checkboxes in its items.

The Devil is in the Details

This sounds too good to be true, and it is. Making the TreeView “feel right,” from a keyboard navigation perspective, is not quite as simple. The fundamental problem is that as you navigate the tree via arrow keys, a TreeViewItem will first take input focus, and then the CheckBox it contains will take focus upon the next keystroke. Both the TreeViewItem and CheckBox controls are focusable. The result is that you must press an arrow key twice to navigate from item to item in the tree. That is definitely not an acceptable user experience, and there is no simple property that you can set to make it work properly. I have already brought this issue to the attention of a certain key member on the WPF team at Microsoft, so they might address it in a future version of the platform.

Functional Requirements

Before we start to examine how this demo program works, first we will review what it does. Here is a screenshot of the demo application in action:

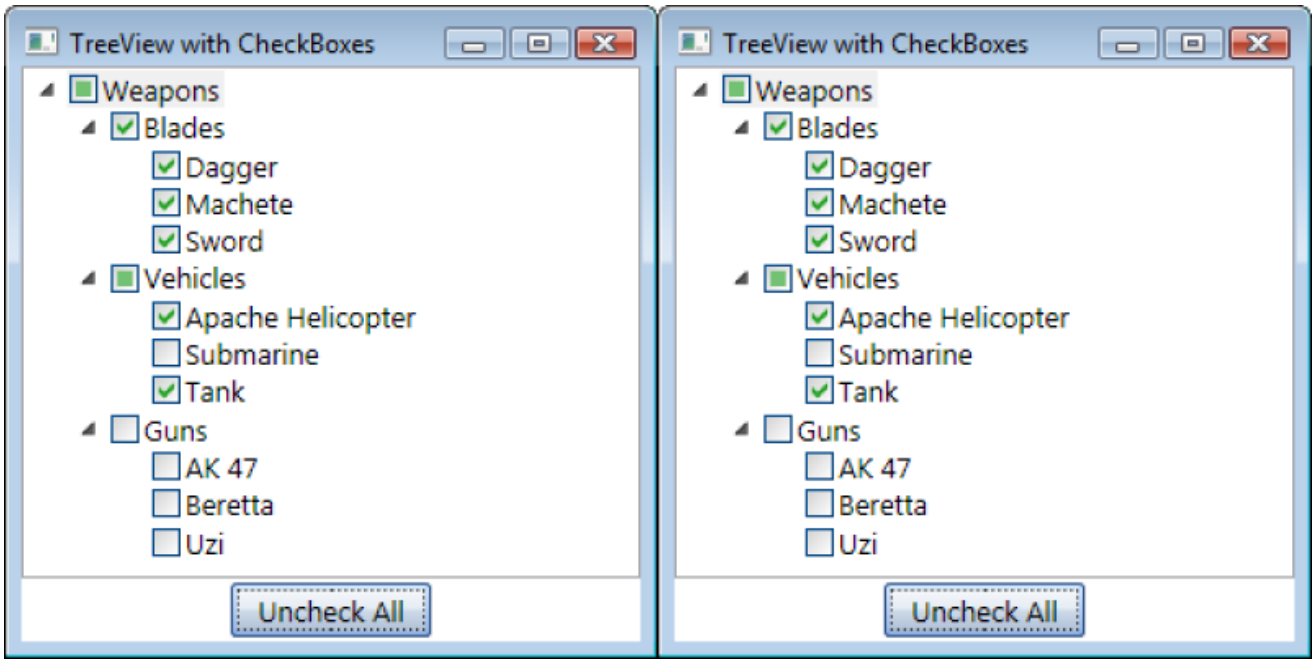
Datenbindung zwei Kernaspekte von WPF sind, bietet das WPF TreeView keine integrierte Unterstützung für die Anzeige von Kontrollkästchen. Es ist sehr einfach, ein CheckBox-Steuerelement im ItemTemplate eines TreeViews zu deklarieren und plötzlich enthält jedes Element im Baum eine CheckBox. Fügen Sie der IsChecked-Eigenschaft einen einfachen {Binding}-Ausdruck hinzu, und plötzlich ist der Prüfstatus dieser Boxen an eine Eigenschaft der zugrunde liegenden Datenobjekte gebunden. Es wäre bestenfalls überflüssig, dass die WPF TreeView eine API speziell für die Anzeige von Checkboxen in ihren Elementen hat.

Der Teufel steckt im Detail

Das klingt zu schön, um wahr zu sein, und das ist es auch. Den TreeView aus der Perspektive der Tastaturnavigation „richtig“ zu machen, ist nicht ganz so einfach. Das grundsätzliche Problem ist, dass ein TreeViewItem beim Navigieren durch den Baum mit den Pfeiltasten zuerst den Eingabefokus erhält und dann die CheckBox, die es enthält, beim nächsten Tastendruck den Fokus erhält. Sowohl das TreeViewItem- als auch das CheckBox-Steuerelement sind fokussierbar. Das Ergebnis ist, dass Sie eine Pfeiltaste zweimal drücken müssen, um im Baum von einem Element zum anderen zu navigieren. Das ist definitiv keine akzeptable Benutzererfahrung, und es gibt keine einfache Eigenschaft, die Sie einstellen können, damit es richtig funktioniert. Ich habe bereits ein bestimmtes Mitglied des WPF-Teams bei Microsoft auf dieses Problem aufmerksam gemacht, damit es in einer zukünftigen Version der Plattform behoben werden kann.

Funktionale Anforderungen

Bevor wir untersuchen, wie dieses Demoprogramm funktioniert, sollten wir uns zunächst ansehen, was es tut. Hier ist ein Screenshot der Demoanwendung in Aktion:



Now let's see what the functional requirements are:

1. Requirement : Each item in the tree must display a checkbox that displays the text and check state of an underlying data object.
2. Requirement : Upon an item being checked or unchecked, all of its child items should be checked or unchecked, respectively.
3. Requirement : If an item's descendants do not all have the same check state, that item's check state must be 'indeterminate.'
4. Requirement : Navigating from item to item should require only one press of an arrow key.
5. Requirement : Pressing the Spacebar or Enter keys should toggle the check state of the selected item.
6. Requirement : Clicking on an item's checkbox should toggle its check state, but not select the item.
7. Requirement : Clicking on an item's display text should select the item, but not toggle its check state.
8. Requirement : All items in the tree should be in the expanded state by default.

I suggest you copy those requirements and paste them into your favorite text editor, such as Notepad, because we will reference them

Schauen wir uns nun die funktionalen Anforderungen an:

1. Anforderung : Jedes Element in der Baumstruktur muss ein Kontrollkästchen enthalten, das den Text und den Kontrollstatus eines zugrunde liegenden Datenobjekts anzeigt.
2. Anforderung : Wenn ein Element angekreuzt oder nicht angekreuzt wird, sollten alle seine untergeordneten Elemente angekreuzt bzw. nicht angekreuzt werden.
3. Anforderung : Wenn die Nachkommen eines Eintrags nicht alle den gleichen Prüfstatus haben, muss der Prüfstatus dieses Eintrags „unbestimmt" sein.
4. Anforderung : Das Navigieren von Element zu Element sollte nur einen einzigen Druck auf eine Pfeiltaste erfordern.
5. Anforderung : Das Drücken der Leertaste oder der Eingabetaste sollte den Prüfstatus des ausgewählten Eintrags umschalten.
6. Anforderung : Ein Klick auf das Kontrollkästchen eines Eintrags soll den Kontrollstatus umschalten, aber den Eintrag nicht auswählen.
7. Anforderung : Das Anklicken des Anzeigetextes eines Eintrags soll den Eintrag auswählen, aber nicht seinen Markierungsstatus umschalten.
8. Anforderung : Alle Elemente in der

throughout the rest of the article by number.

Putting the Smarts in a ViewModel

As explained in my 'Simplifying the WPF TreeView by Using the ViewModel Pattern' article, the TreeView was practically designed to be used in conjunction with a ViewModel. This article takes that idea further, and shows how we can use a ViewModel to encapsulate application-specific logic related to the check state of items in the tree. In this article, we will examine my FooViewModel class, which the following interface describes:

```
1. interface IFooViewModel :
   INotifyPropertyChanged
2. {
3.     List<FooViewModel> Children
   { get; }
4.     bool? IsChecked { get; set;
   }
5.     bool IsInitiallySelected {
   get; }
6.     string Name { get; }
7. }
```

The most interesting aspect of this ViewModel class is the logic behind the IsChecked property. This logic satisfies Requirements 2 and 3, seen previously. The FooViewModel's IsChecked logic is below:

```
1. /// <summary>
2. /// Gets/sets the state of the
   associated UI toggle (ex.
   CheckBox).
3. /// The return value is
   calculated based on the check
   state of all
4. /// child FooViewModels.
   Setting this property to true
   or false
5. /// will set all children to
   the same check state, and
   setting it
6. /// to any value will cause the
   parent to verify its check
   state.
```

Baumstruktur sollten sich standardmäßig im erweiterten Zustand befinden.

Ich schlage vor, Sie kopieren diese Anforderungen und fügen sie in Ihren bevorzugten Texteditor ein, z. B. in Notepad, da wir sie im weiteren Verlauf des Artikels nummerisch referenzieren werden.

Die Intelligenz in ein ViewModel packen

Wie in meinem Artikel 'Simplifying the WPF TreeView by Using the ViewModel Pattern' (Vereinfachung der WPF-TreeView durch Verwendung des ViewModel-Musters) erläutert, wurde die TreeView praktisch dafür entwickelt, in Verbindung mit einem ViewModel verwendet zu werden. Dieser Artikel führt diese Idee weiter und zeigt, wie wir ein ViewModel verwenden können, um anwendungsspezifische Logik in Bezug auf den Prüfstatus von Elementen im Baum zu kapseln. In diesem Artikel werden wir meine FooViewModel-Klasse untersuchen, die durch die folgende Schnittstelle beschrieben wird:

```
1. interface IFooViewModel :
   INotifyPropertyChanged
2. {
3.     List<FooViewModel> Children
   { get; }
4.     bool? IsChecked { get; set;
   }
5.     bool IsInitiallySelected {
   get; }
6.     string Name { get; }
7. }
```

```
1. ///Ruft den Zustand des
   zugehörigen UI-Toggles (z.B.
   CheckBox) ab bzw. setzt ihn.
2. ///Der Rückgabewert wird auf
   der Grundlage des Prüfstatus
   aller untergeordneten
3. ///FooViewModelle berechnet.
   Wenn diese Eigenschaft auf true
   oder false gesetzt wird,
4. ///erhalten alle
   untergeordneten Modelle den
```

```
/// </summary>
public bool? IsChecked
{
    get { return _isChecked; }
    set { this.SetIsChecked(value, true, true); }
}

void SetIsChecked(bool? value, bool updateChildren, bool updateParent)
{
    if (value == _isChecked)
        return;

    _isChecked = value;

    if (updateChildren && _isChecked.HasValue)
        this.Children.ForEach(c => c.SetIsChecked(_isChecked, true, false));

    if (updateParent && _parent != null)
        _parent.VerifyCheckState();

    this.OnPropertyChanged("IsChecked");
}

void VerifyCheckState()
{
    bool? state = null;
    for (int i = 0; i < this.Children.Count; ++i)
    {
        bool? current = this.Children[i].IsChecked;
        if (i == 0)
        {
            state = current;
        }
        else if (state != current)
        {
            state = null;
```

```
    gleichen Prüfstatus,
///und wenn sie auf einen beliebigen Wert gesetzt wird, überprüft das übergeordnete Modell seinen Prüfstatus.
public bool? IsChecked
{
    get { return _isChecked; }
    set { this.SetIsChecked(value, true, true); }
}

void SetIsChecked(bool? value, bool updateChildren, bool updateParent)
{
    if (value == _isChecked)
        return;

    _isChecked = value;

    if (updateChildren && _isChecked.HasValue)
        this.Children.ForEach(c => c.SetIsChecked(_isChecked, true, false));

    if (updateParent && _parent != null)
        _parent.VerifyCheckState();

    this.OnPropertyChanged("IsChecked");
}

void VerifyCheckState()
{
    bool? state = null;
    for (int i = 0; i < this.Children.Count; ++i)
    {
        bool? current = this.Children[i].IsChecked;
        if (i == 0)
        {
            state = current;
        }
```

```
            break;
        }
    }
    this.SetIsChecked(state, false, true);
}
```

This strategy is specific to the functional requirements I imposed upon myself. If you have different rules regarding how and when items should update their check state, simply adjust the logic in those methods to suit your needs.

TreeView Configuration

Now it is time to see how the TreeView is able to display checkboxes and bind to the ViewModel. This is entirely accomplished in XAML. The TreeView declaration is actually quite simple, as seen below:

```
<TreeView
  x:Name="tree"
  ItemContainerStyle="{StaticResource TreeViewItemStyle}"
  ItemsSource="{Binding Mode=OneTime}"
  ItemTemplate="{StaticResource CheckBoxItemTemplate}"
  />
</code
The TreeView's ItemsSource property is implicitly bound to its DataContext, which inherits a List<FooViewModel> from the containing window. That list only contains one ViewModel object, but it is necessary to put it into a collection because ItemsSource is of type IEnumerable.

TreeViewItem is a container of visual elements generated by the ItemTemplate. In this demo, we assign the following HierarchicalDataTemplate to the tree's ItemTemplate property:

```

```
        else if (state != current)
        {
            state = null;
            break;
        }
    }
    this.SetIsChecked(state, false, true);
}
```

```
<code C# [enable_line_numbers="true",highlight_lines_extra="0,"]>
<HierarchicalDataTemplate
  x:Key="CheckBoxItemTemplate"
  ItemsSource="{Binding Children, Mode=OneTime}"
  >
  <StackPanel Orientation="Horizontal">
    <!-- These elements are bound to a FooViewModel object. -->
    <CheckBox
      Focusable="False"
      IsChecked="{Binding IsChecked}"
      VerticalAlignment="Center"
      />
    <ContentPresenter
      Content="{Binding Name, Mode=OneTime}"
      Margin="2,0"
      />
  </StackPanel>
</HierarchicalDataTemplate>
```

There are several points of interest in that template. The template includes a CheckBox whose Focusable property is set to false. This prevents the CheckBox from ever receiving input focus, which assists in meeting Requirement 4. You might be wondering how we will be able to satisfy Requirement 5 if the CheckBox never has input focus. We will address that issue later in this article, when we examine how to attach the behavior of a ToggleButton to a TreeViewItem.

The CheckBox’s IsChecked property is bound to the IsChecked property of a FooViewModel object, but notice that its Content property is not set to anything. Instead, there is a ContentPresenter directly next to it, whose Content is bound to the Name property of a FooViewModel object. By default, clicking anywhere on a CheckBox causes it to toggle its check state. By using a separate ContentPresenter, rather than setting the CheckBox’s Content property, we can avoid that

default behavior. This helps us satisfy Requirements 6 and 7. Clicking on the box element in the CheckBox will cause its check state to change, but clicking on the neighboring display text will not. Similarly, clicking on the box in the CheckBox will not select that item, but clicking on the neighboring display text will.

We will examine the TreeView's ItemContainerStyle in the next section.

Turning a TreeViewItem into a ToggleButton

In the previous section, we quickly considered an interesting question. If the CheckBox in the TreeViewItem has its Focusable property set to false, how can it toggle its check state in response to the Spacebar or Enter key? Since an element only receives keystrokes if it has keyboard focus, it seems impossible for Requirement 5 to be satisfied. Keep in mind; we had to set the CheckBox's Focusable property to false so that navigating from item to item in the tree does not require multiple keystrokes.

This is a tricky problem: we cannot let the CheckBox ever have input focus because it negatively affects keyboard navigation, yet, when its containing item is selected, it must somehow toggle its check state in response to certain keystrokes. These seem to be mutually exclusive requirements. When I hit this brick wall, I decided to seek geek from the WPF Disciples, and started this thread. Not to my surprise, Dr. WPF had already encountered this type of problem and devised a brilliant-approaching-genius solution that was easy to plug into my application. The good Doctor sent me the code for a VirtualToggleButton class, and was kind enough to allow me to publish it in this article.

The Doctor's solution uses what John Gossman refers to as "attached behavior." The idea is that you set an attached property on an element so that you can gain access to the element from the class that exposes the attached property. Once that class has access to the element, it can hook events on it and, in response to those events firing, make the element do things that it normally would not do. It is a very convenient

alternative to creating and using subclasses, and is very XAML-friendly.

In this article, we see how to give a TreeViewItem an attached IsChecked property that toggles when the user presses the Spacebar or Enter key. That attached IsChecked property binds to the IsChecked property of a FooViewModel object, which is also bound to the IsChecked property of the CheckBox in the TreeViewItem. This solution gives the appearance that a CheckBox is toggling its check state in response to the Spacebar or Enter key, but in reality, its IsChecked property updates in response to a TreeViewItem pushing a new value to the ViewModel's IsChecked property via data binding.

Before going any further, I should point out that I fully recognize that this is crazy. The fact that this is the cleanest way to implement a TreeView of checkboxes in WPF v3.5 indicates, to me, that Microsoft needs to simplify this aspect of the platform. However, until they do, this is probably the best way to implement the feature.

In this demo, we do not make use of all features in Dr. WPF's VirtualToggleButton class. It has support for several things that we do not need, such as handling mouse clicks and providing tri-state checkboxes. We only need to make use of its support for the attached IsVirtualToggleButton and IsChecked properties and the keyboard interaction behavior it provides.

Here is the property-changed callback method for the attached IsVirtualToggleButton property, which is what enables this class to gain access to TreeViewItems in the tree:

```
1. /// <summary>
2. /// Handles changes to the
   IsVirtualToggleButton property.
3. /// </summary>
4. private static void
   OnIsVirtualToggleButtonChanged(
5.   DependencyObject d,
   DependencyPropertyChangedEventA
   rgs e)
```

```
 6. {
 7.     IInputElement element = d
    as IInputElement;
 8.     if (element != null)
 9.     {
10.         if ((bool)e.NewValue)
11.         {
12.
    element.MouseLeftButtonDown +=
    OnMouseLeftButtonDown;
13.             element.KeyDown +=
    OnKeyDown;
14.         }
15.         else
16.         {
17.
    element.MouseLeftButtonDown -=
    OnMouseLeftButtonDown;
18.             element.KeyDown -=
    OnKeyDown;
19.         }
20.     }
21. }
```

When a TreeViewItem raises its KeyDown event, this logic executes:

```
 1. private static void
    OnKeyDown(object sender,
    KeyEventArgs e)
 2. {
 3.     if (e.OriginalSource ==
    sender)
 4.     {
 5.         if (e.Key == Key.Space)
 6.         {
 7.             // ignore alt+space
    which invokes the system menu
 8.             if
    ((Keyboard.Modifiers &
    ModifierKeys.Alt) ==
    ModifierKeys.Alt)
 9.                 return;
10.
11.
    UpdateIsChecked(sender as
    DependencyObject);
12.             e.Handled = true;
13.         }
14.         else if (e.Key ==
```

```
    Key.Enter &&
15.          (bool)(sender as
    DependencyObject)
16.
    .GetValue(KeyboardNavigation.Ac
    ceptsReturnProperty))
17.         {
18.
    UpdateIsChecked(sender as
    DependencyObject);
19.             e.Handled = true;
20.         }
21.     }
22. }
23.
24. private static void
    UpdateIsChecked(DependencyObjec
    t d)
25. {
26.     Nullable<bool> isChecked =
    GetIsChecked(d);
27.     if (isChecked == true)
28.     {
29.         SetIsChecked(d,
30.          GetIsThreeState(d) ?
31.          (Nullable<bool>)null :
32.
    (Nullable<bool>)false);
33.     }
34.     else
35.     {
36.         SetIsChecked(d,
    isChecked.HasValue);
37.     }
38. }
```

The UpdateIsChecked method sets the attached IsChecked property on an element, which is a TreeViewItem in this demo. Setting an attached property on a TreeViewItem has no effect by itself. In order to have the application use that property value, it must be bound to something. In this application, it is bound to the IsChecked property of a FooViewModel object. The following Style is assigned to the TreeView's ItemContainerStyle property. It ties a TreeViewItem to a FooViewModel object and adds the virtual ToggleButton behavior that we just examined.

```
1. <Style
   x:Key="TreeViewItemStyle"
   TargetType="TreeViewItem">
2.   <Setter Property="IsExpanded"
   Value="True" />
3.   <Setter Property="IsSelected"
   Value="{Binding
   IsInitiallySelected,
   Mode=OneTime}" />
4.   <Setter
   Property="KeyboardNavigation.Ac
   ceptsReturn" Value="True" />
5.   <Setter
   Property="dw:VirtualToggleButto
   n.IsVirtualToggleButton"
   Value="True" />
6.   <Setter
   Property="dw:VirtualToggleButto
   n.IsChecked" Value="{Binding
   IsChecked}" />
7. </Style>
```

This piece ties the entire puzzle together. Note that the attached KeyboardNavigation.AcceptsReturn property is set to true on each TreeViewItem so that the VirtualToggleButton will toggle its check state in response to the Enter key. The first Setter in the Style, which sets the initial value of each item's IsExpanded property to true, ensures that Requirement 8 is met.

CheckBox Bug in Aero Theme

I must point out one strange, and disappointing, issue. The Aero theme for WPF's CheckBox control has a problem in .NET 3.5. When it moves from the 'Indeterminate' state to the 'Checked' state, the background of the box does not update properly until you move the mouse cursor over it. You can see this in the screenshot below: